KNITRO™ 7.0

User's Manual

KNITRO User's Manual

Version 7.0

Richard A. Waltz
Todd D. Plantenga
Ziena Optimization, Inc.
www.ziena.com

September 2010

©2004-2010 Ziena Optimization, Inc.

Contents

Contents a

1 Introduction 1
- 1.1 Product overview . . . 1
- 1.2 Algorithms overview . . . 2
- 1.3 What's new in version 7.0 . . . 3
- 1.4 Contact and support information . . . 4

2 Installation 5
- 2.1 Windows . . . 5
- 2.2 Unix (Linux, Mac OS X, Solaris) . . . 7
- 2.3 Linux compatibility issues . . . 9
- 2.4 Troubleshooting . . . 10

3 Using KNITRO with the AMPL modeling language 11
- 3.1 Example AMPL model solution using KNITRO . . . 11
- 3.2 KNITRO options for AMPL . . . 15
- 3.3 Solving with complementarity constraints . . . 21
- 3.4 Specifying MIP branching priorities . . . 21
- 3.5 Displaying AMPL variables in KNITRO . . . 22

4 The KNITRO callable library 23
- 4.1 KNITRO in a C application . . . 23
- 4.2 Example in C using callback mode . . . 30
- 4.3 Example in C using reverse communication mode . . . 35
- 4.4 KNITRO in a C++ application . . . 38
- 4.5 KNITRO in a Java application . . . 38
- 4.6 KNITRO in a Fortran application . . . 39
- 4.7 Compiler specifications . . . 39
- 4.8 Specifying the Jacobian and Hessian matrices . . . 40
- 4.9 Determining convexity . . . 42
- 4.10 Calling without first derivatives . . . 42

5 User options in KNITRO 45
- 5.1 Description of KNITRO user options . . . 45
- 5.2 The KNITRO options file . . . 60
- 5.3 Setting options through function calls . . . 61
- 5.4 Loading dynamic libraries . . . 61

6 KNITRO termination test and optimality 62
- 6.1 Continuous problems . . . 62
- 6.2 Discrete or mixed integer problems . . . 63

7 KNITRO output and solution information **65**
7.1 Understanding KNITRO output for continuous problems . . . 65
7.2 Understanding KNITRO output for discrete problems . . . 68
7.3 Accessing solution information . . . 71

8 Algorithm options **74**
8.1 Automatic . . . 74
8.2 Interior/Direct . . . 74
8.3 Interior/CG . . . 74
8.4 Active Set . . . 74

9 Other KNITRO special features **75**
9.1 First derivative and gradient check options . . . 75
9.2 Second derivative options . . . 75
9.3 Feasibility options . . . 77
9.4 Honor bounds mode . . . 77
9.5 Crossover . . . 77
9.6 Multi-start . . . 78
9.7 Reverse communication mode for invoking KNITRO . . . 79
9.8 Callback mode for invoking KNITRO . . . 80

10 Special problem classes **82**
10.1 Linear programming problems (LPs) . . . 82
10.2 Quadratic programming problems (QPs) . . . 82
10.3 Systems of nonlinear equations . . . 82
10.4 Least squares problems . . . 82
10.5 Complementarity constraints (MPCCs) . . . 83
10.6 Global optimization . . . 86
10.7 Mixed integer programming (MIP) . . . 87

11 Tips and troubleshooting **88**
11.1 Option tuning for efficiency . . . 88
11.2 Memory issues . . . 89
11.3 Infeasibility . . . 89

References **90**

Appendix A Solution Status Codes **91**

Appendix B List of Output Files **95**

1 Introduction

This chapter gives an overview of the KNITRO optimization software package, and details concerning contact and support information.

1.1 Product overview

KNITRO 7.0 is an optimization software library for finding solutions of both continuous (smooth) optimization models (with or without constraints), as well as discrete optimization models with integer or binary variables (i.e. mixed integer programs). KNITRO is primarily designed for finding local solutions of large-scale, continuous nonlinear problems. Multi-start heuristics are provided for trying to locate the global solution. Although primarily designed for general, nonlinear optimization, KNITRO is efficient at solving all of the following classes of optimization problems:

- unconstrained
- bound constrained
- systems of nonlinear equations
- least squares problems, both linear and nonlinear
- linear programming problems (LPs)
- quadratic programming problems (QPs), both convex and nonconvex
- mathematical programs with complementarity constraints (MPCCs)
- general nonlinear (smooth) constrained problems (NLP), both convex and nonconvex
- mixed integer linear programs (MILP) of moderate size
- mixed integer (convex) nonlinear programs (MINLP) of moderate size

The KNITRO package provides the following features:

- efficient and robust solution of small or large problems
- solvers for both continuous and discrete problems
- derivative-free, 1st derivative, and 2nd derivative options
- option to remain feasible throughout the optimization or not
- both interior-point (barrier) and active-set methods
- both iterative and direct approaches for computing steps
- support for Windows (32-bit and 64-bit), Linux (32-bit and 64-bit), Mac OS X (32-bit and 64-bit) and Solaris 10 (64-bit, x86_64).
- programmatic interfaces: C/C++, Fortran, Java, Microsoft Excel
- modeling language interfaces: AMPL®, AIMMS®, GAMS®, *Mathematica*®, MATLAB®, MPL
- thread-safe libraries for easy embedding into application software

1.2 Algorithms overview

The problems solved by KNITRO have the form

$$\underset{x}{\text{minimize}} \quad f(x) \tag{1.1a}$$

$$\text{subject to} \quad c^L \leq c(x) \leq c^U \tag{1.1b}$$

$$b^L \leq x \leq b^U, \tag{1.1c}$$

where $x \in \mathbf{R}^n$ are the unknown variables (which can be specified as continuous, binary or integer), c^L and c^U are lower and upper bounds (possibly infinite) on the general constraints, and b^L and b^U are lower and upper simple bounds (possibly infinite) on the variables. This formulation allows many types of constraints, including equalities (if $c^L = c^U$), fixed variables (if $b^L = b^U$), and both single and double-sided inequality constraints or bounded variables. Complementarity constraints may also be included (see Section 10.5). KNITRO assumes that the functions $f(x)$, and $c(x)$ are smooth, although problems with derivative discontinuities can often be solved successfully.

KNITRO implements three state-of-the-art interior-point and active-set methods for solving continuous, nonlinear optimization problems. Each algorithm possesses strong convergence properties and is coded for maximum efficiency and robustness. However, the algorithms have fundamental differences that lead to different behavior on nonlinear optimization problems. Together, the three methods provide a suite of different ways to attack difficult problems.

We encourage the user to try all algorithmic options to determine which one is more suitable for the application at hand. For guidance on choosing the best algorithm see Section 8.

Interior/Direct algorithm: Interior-point methods (also known as barrier methods) replace the nonlinear programming problem by a series of barrier subproblems controlled by a barrier parameter μ. Trust regions and a merit function are used to promote convergence. Interior-point methods perform one or more minimization steps on each barrier subproblem, then decrease the barrier parameter and repeat the process until the original problem (1.1) has been solved to the desired accuracy. The Interior/Direct method computes new iterates by solving the primal-dual KKT matrix using direct linear algebra. The method may temporarily switch to the Interior/CG algorithm if it encounters difficulties.

Interior/CG algorithm: This method is similar to the Interior/Direct algorithm, except the primal-dual KKT system is solved using a projected conjugate gradient iteration. This approach differs from most interior-point methods proposed in the literature. A projection matrix is factorized and conjugate gradient applied to approximately minimize a quadratic model of the barrier problem. The use of conjugate gradient on large-scale problems allows KNITRO to utilize exact second derivatives without forming the Hessian matrix.

Active Set algorithm: Active set methods solve a sequence of subproblems based on a quadratic model of the original problem. In contrast with interior-point methods, the algorithm seeks active inequalities and follows a more exterior path to the solution. KNITRO implements a sequential linear-quadratic programming (SLQP) algorithm, similar in nature to a sequential quadratic programming method but using linear programming subproblems to estimate the active set. This method may be preferable to interior-point algorithms when a good initial point can be provided; for example, when solving a sequence of related problems. KNITRO can also "crossover" from an interior-point method and apply Active Set to provide highly accurate active set and sensitivity information (see Section 9.5).

For mixed integer programs (MIPs), KNITRO provides two variants of the branch and bound algorithm. The first is a standard implementation, while the second is specialized for convex, mixed integer nonlinear problems.

For a detailed description of the algorithm implemented in Interior/CG see [4] and for the global convergence theory see [1]. The method implemented in Interior/Direct is described in [11]. The Active Set algorithm is described in [3] and the global convergence theory for this algorithm is in [2]. A summary of the algorithms and techniques implemented in the KNITRO software product is given in [6]. To solve linear systems arising at every iteration of the algorithm, KNITRO may utilize routines `MA27` or `MA57` [8], a component package of the Harwell Subroutine Library (www.cse.clrc.ac.uk/activity/HSL). In addition, the Active Set algorithm in KNITRO may make use of the COIN-OR Clp linear programming solver module. The version used in KNITRO may be downloaded from http://www.ziena.com/clp.html.

1.3 What's new in version 7.0

- KNITRO 7.0 introduces several improvements in linear algebra to improve performance. In addition to the HSL `MA27` routine, KNITRO now also offers the option to use the HSL `MA57` routine to solve sparse linear systems. `MA57` will often be faster on very large problems. In addition, since `MA57` uses Basic Linear Algebra Subroutines (BLAS), it can more effectively exploit optimized BLAS and parallel BLAS routines for speedups.

- KNITRO 7.0 offers a new, dense QR option for solving linear systems. This option is restricted to small problems, but may often be the most effective when the Jacobian and/or Hessian matrix is nearly dense. The dense QR option is based on the QR routines provided in LAPACK.

- The new options for solving linear systems in KNITRO 7.0 are controlled by the new `linsolver` user option. By default, KNITRO tries to determine the best solver to use. See Section 5.1 for more details on this option.

- The new user option `bar_directinterval` has been added to control the maximum number of iterations KNITRO will go before trying to enforce that a step is taken using direct linear algebra (as opposed to using a conjugate gradient step). This option is only valid for the Interior/Direct algorithm and may be useful on problems where KNITRO appears to be taking lots of conjugate gradient (CG) steps. Setting `bar_directinterval=0` will try to enforce that only direct steps are taken. See Section 5.1 for more details.

- A new API function `KTR_mip_set_branching_priorities()` was added to allow users to specify a branching priority for integer variables when solving mixed integer programs in KNITRO 7.0. See Section 4.1 for more details.

- Improved support for 64-bit architectures, including new support for Intel-based 64-bit Mac OS X and 64-bit Sun Solaris on x86_64. See 1.4 for a full list of supported platforms.

- Improvements in quickly identifying infeasible models have been made in KNITRO 7.0, particularly for the active-set algorithm. A new user options `infeastol` has been added for declaring infeasibility. See Section 5.1 for more details.

- Significant speedups have been made when using the LBFGS limited-memory quasi-Newton approximation for 2nd derivatives.

- General performance improvements have been made for both the interior-point/barrier solvers in KNITRO 7.0 to improve speed and robustness.

1.4 Contact and support information

KNITRO 7.0 is licensed by Ziena Optimization, Inc. (http://www.ziena.com/), and is supported on the platforms described in the table below. Note that the Solaris platform is only currently supported for single-machine licensing.

PLATFORM	OPERATING SYSTEM	PROCESSOR
Windows 32-bit	Windows XP, Windows Server 2003, Windows Server 2008, Vista, Windows 7	AMD Duron/Intel Pentium3 or later x86 CPU
Windows 64-bit	Windows XP, Windows Server 2003, Windows Server 2008, Vista, Windows 7	Any AMD64 or Intel EM64T enabled 64bit CPU
Linux 32-bit	RedHat (glibc2.3.3+) compatible	AMD Duron/Intel Pentium3 or later x86 CPU
Linux 64-bit	RedHat (glibc2.3.4+) compatible	Any AMD64 or Intel EM64T enabled 64bit CPU
Mac OS X 32-bit	Version 10.4 (tiger) or later	Intel x86 CPU
Mac OS X 64-bit	Version 10.5 (leopard) or later	Intel EM64T enabled 64bit CPU
Solaris 64-bit	Solaris 10 (single-machine licensing only)	Any AMD64 or Intel EM64T enabled 64bit CPU

General information regarding KNITRO can be found at the KNITRO website:

http://www.ziena.com/knitro.html

For technical support, licensing, or general information send email to:

info@ziena.com

or contact your local distributor.

2 Installation

Instructions for installing the KNITRO package on supported platforms are given below. After installing, view the INSTALL.txt, LICENSE_KNITRO.txt, and README.txt files, then test the installation.

If you purchased the KNITRO/AMPL solver product, then refer to Section 3 and test KNITRO as the solver for any smooth optimization model (an AMPL test model is provided with the KNITRO distribution).

If you purchased the full KNITRO product, then test KNITRO by compiling and running one or more programs in the examples directory. Example problems are provided for C, C++, Fortran, and Java interfaces. We recommend understanding these examples and reading Section 4 of this manual before proceeding with development of your own application interface.

2.1 Windows

KNITRO is supported on Windows 2003, Windows XP SP2, Windows XP Professional x64, Windows Vista and Windows 7. There are compatibility problems with Windows XP SP1 system libraries – users should upgrade to Windows XP SP2. The KNITRO software package for Windows is delivered as a zipped file ending in .zip, or as a self-extracting executable ending in .exe. For the .zip file, double-click on it and extract all contents to a new folder. For the .exe file, double-click on it and follow the instructions. The self-extracting executable creates start menu shortcuts and an uninstall entry in Add/Remove Programs; otherwise, the two install methods are identical.

The default installation location for KNITRO is (assuming your `%HOMEDRIVE%` is `C:`)

```
C:\Program Files\Ziena\
```

Unpacking will create a folder named knitro-7.x-z (or knitroampl-7.x-z for the KNITRO/AMPL solver product). Contents of the full product distribution are the following:

INSTALL.txt:	A file containing installation instructions.
LICENSE_KNITRO.txt:	A file containing the KNITRO license agreement.
README.txt:	A file with instructions on how to get started using KNITRO.
KNITRO70-ReleaseNotes.txt:	A file containing release notes.
get_machine_ID:	An executable that identifies the machine ID, required for obtaining a Ziena license file.
doc:	A folder containing KNITRO documentation, including this manual.
include:	A folder containing the KNITRO header file knitro.h.
lib:	A folder containing the KNITRO library and object files: knitro_objlib.a, knitro.lib and knitro.dll.
examples:	A folder containing examples of how to use the KNITRO API in different programming languages (C, C++, Fortran, Java). The examples\C folder contains the most extensive set (see examples\C\README.txt for details).

knitroampl: A folder containing knitroampl.exe (the KNITRO solver for AMPL), instructions, and an example model for testing KNITRO with AMPL.

To activate KNITRO for your computer you will need a valid Ziena license file. If you purchased a floating network license, then refer to the *Ziena License Manager User's Manual* provided in the `doc` folder of your distribution. For a stand-alone, single computer license, double-click on the `get_machine_ID.bat` batch file provided with the distribution. This will generate a machine ID (five pairs of hexadecimal digits). Alternatively, open a DOS-like command window (click Start → Run, and then type `cmd`). Change to the directory where you unzipped the distribution, and type `get_machine_ID.exe`, a program supplied with the distribution to generate the machine ID.

Email the machine ID to info@ziena.com if purchased through Ziena. (If KNITRO was purchased through a distributor, then email the machine ID to your local distributor.) Ziena (or your local distributor) will then send a license file containing the encrypted license text string. The Ziena license manager supports a variety of ways to install licenses. The simplest procedure is to copy each license into a file whose name begins with the characters `"ziena_lic"`. Then place the file in the folder

```
C:\Program Files\Ziena\
```

If this does not work, try creating a new environment variable called `ZIENA_LICENSE` and set it to the folder holding your license file(s). See information on setting environment variables below and refer to the *Ziena License Manager User's Manual* for more installation details.

Setting environment variables

In order to run KNITRO binary or executable files from anywhere on your Windows computer, as well as load dynamic libraries (or dll's) used by KNITRO at runtime, it is necessary to make sure that the `Path` system environment variable is set properly on your Windows machine. In particular, you must update the system `Path` environment variable so that it indicates the location of the KNITRO `lib` folder (containing the KNITRO provided dll's) and the `knitroampl` folder (or whichever folder contains the `knitroampl.exe` executable file). This can be done as follows.

Windows Vista and Windows 7

1. At the Windows desktop, right-click `Computer`.
2. Select `Properties`.
3. Click on `Advanced System Settings` in the left pane.
4. In the `System Properties` window select the `Advanced` tab.
5. Click `Environment Variables`.
6. Under `System variables`, edit the `Path` variable to add the KNITRO `lib` folder and `knitroampl` folder. Specify the whole path to these folders, and make sure to separate the paths by a semicolon.

Windows XP

1. At the Windows desktop, right-click `My Computer`.
2. Select `Properties`.
3. Click the `Advanced` tab.

4. Click `Environment Variables`.
5. Under `System variables`, edit the `Path` variable to add the KNITRO `lib` folder and `knitroampl` folder. Specify the whole path to these folders, and make sure to separate the paths by a semicolon.

Note that you may need to restart your Windows machine after modifying the environment variables, for the changes to take effect. Simply logging out and relogging in is not enough. Moreover, if the `Path` environment variable points to more than one folder that contains an executable or dll of the same name, the one that will be chosen is the one whose folder appears first in the `Path` variable definition.

If you are using KNITRO with AMPL, you should make sure the folder containing the AMPL executable file `ampl.exe` is also added to the `Path` variable (as well as the folder containing the `knitroampl.exe` as described above). Additionally, if you are using an external third party dll with KNITRO such as your own Basic Linear Algebra Subroutine (BLAS) libraries (see user options `blasoption` and `blasoptionlib` in Section 5.1), or a Cplex® library (see user option `lpsolver` in Section 5.1), then you will also need to add the folders containing these dll's to the system `Path` environment variable as described in the last step above.

If you are setting the `ZIENA_LICENSE` environment variable to activate your license, then follow the instructions above, but in the last step create a new environment variable called `ZIENA_LICENSE` and give it the value of the folder containing your Ziena license file (specify the whole path to this folder). For more installation options and general troubleshooting, see Section 2.4 and read the *Ziena License Manager User's Manual*.

2.2 Unix (Linux, Mac OS X, Solaris)

KNITRO is supported on Linux (32-bit and 64-bit, all distributions), Mac OS X (32-bit x86 on Mac OS X 10.4 or higher and 64-bit x86_64 on Mac OS X 10.5 or higher), and Solaris 10 (64-bit x86_64, single-machine licenses only).

The KNITRO software package for Unix is delivered as a gzipped tar file. Save this file in a fresh subdirectory on your system. To unpack, type the commands

```
gunzip knitro-7.x-platformname.tar.gz
tar -xvf knitro-7.x-platformname.tar
```

Unpacking will create a directory named knitro-7.x-z (or knitroampl-7.x-z for the KNITRO/AMPL solver product). Contents of the full product distribution are the following:

INSTALL:	A file containing installation instructions.
LICENSE_KNITRO:	A file containing the KNITRO license agreement.
README:	A file with instructions on how to get started using KNITRO.
KNITRO70-ReleaseNotes:	A file containing release notes.
get_machine_ID:	An executable that identifies the machine ID, required for obtaining a Ziena license file.
doc:	A directory containing KNITRO documentation, including this manual.

include:	A directory containing the KNITRO header file knitro.h.
lib:	A directory containing the KNITRO library files: libknitro.a and libknitro.so.
examples:	A directory containing examples of how to use the KNITRO API in different programming languages (C, C++, Fortran, Java). The examples/C directory contains the most extensive set (see examples/C/README.txt for details).
knitroampl:	A directory containing knitroampl (the KNITRO solver for AMPL), instructions, and an example model for testing KNITRO with AMPL.

To activate KNITRO for your computer you will need a valid Ziena license file. If you purchased a floating network license, then refer to the *Ziena License Manager User's Manual*. For a stand-alone license, execute `get_machine_ID`, a program supplied with the distribution. This will generate a machine ID (five pairs of hexadecimal digits). Email the machine ID to info@ziena.com if purchased through Ziena. (If KNITRO was purchased through a distributor, then email the machine ID to your local distributor.) Ziena (or your local distributor) will then send a license file containing the encrypted license text string. Ziena supports a variety of ways to install licenses. The simplest procedure is to copy each license into a file whose name begins with the characters `"ziena_lic"` (please use lower-case letters). Then place the file in your `$HOME` directory.

Setting environment variables

In order to run KNITRO binary or executable files from anywhere on your Unix computer, as well as load dynamic, shared libraries (i.e. `.so` files) used by KNITRO at runtime, it is necessary to make sure that several environment variables are set properly on your machine.

In particular, you must update the `PATH` environment variable so that it indicates the location of the `knitroampl` directory (or whichever directory contains the `knitroampl` executable file). You must also update the `LD_LIBRARY_PATH` (`DYLD_LIBRARY_PATH` on Mac OS X) environment variable so that it indicates the location of the KNITRO `lib` directory (containing the KNITRO provided `.so` shared libraries).

Setting the `PATH` and `LD_LIBRARY_PATH` (`DYLD_LIBRARY_PATH` on Mac OS X) environment variables on Unix systems can be done as follows. In the instructions below, replace `<file_absolute_path>` with the full path to the directory containing the KNITRO binary file (e.g. the `knitroampl` directory), and replace `<file_absolute_library_path>` with the full path to the directory containing the KNITRO shared object library (e.g. the KNITRO `lib` directory).

Linux or Solaris

If you run a Unix bash shell, then type

```
> export PATH=<file_absolute_path>:${PATH}
> export LD_LIBRARY_PATH=<file_absolute_library_path>:${LD_LIBRARY_PATH}
```

If you run a Unix csh or tcsh shell, then type

```
> setenv PATH <file_absolute_path>:${PATH}
> setenv LD_LIBRARY_PATH <file_absolute_library_path>:${LD_LIBRARY_PATH}
```

Mac OS X

Determine the shell being used:
```
echo ${SHELL}
```

If you run a Unix bash shell, then type
```
> export PATH=<file_absolute_path>:${PATH}
> export DYLD_LIBRARY_PATH=<file_absolute_library_path>:${DYLD_LIBRARY_PATH}
```

If you run a Unix csh or tcsh shell, then type
```
> setenv PATH <file_absolute_path>:${PATH}
> setenv DYLD_LIBRARY_PATH <file_absolute_library_path>:${DYLD_LIBRARY_PATH}
```

Note that the value of the environment variable is only valid in the shell in which it was defined. Moreover, if a particular environment variable points to more than one directory that contains a binary or dynamic library of the same name, the one that will be chosen is the one whose directory appears first in the environment variable definition.

If you are using KNITRO with AMPL, you should also make sure the directory containing the AMPL executable file `ampl` is added to the `PATH` environment variable (as well as the directory containing the knitroampl executable file as described above). Additionally, if you are using an external third party runtime library with KNITRO such as your own Basic Linear Algebra Subroutine (BLAS) libraries (see user options `blasoption` and `blasoptionlib` in Section 5.1), or a Cplex® library (see user option `lpsolver` in Section 5.1), then you will also need to add the directories containing these libraries to the `LD_LIBRARY_PATH` (`DYLD_LIBRARY_PATH` on Mac OS X) environment variable.

If you are setting the `ZIENA_LICENSE` environment variable to activate your license, then follow the instructions above to create a new environment variable called `ZIENA_LICENSE` and give it the value of the directory containing your Ziena license file (specify the whole path to this directory). For more installation options and general troubleshooting, see Section 2.4 and read the *Ziena License Manager User's Manual*.

2.3 Linux compatibility issues

Linux platforms sometimes generate link errors when building the programs in examples/C. Simply type `"gmake"` and see if the build is successful. You may see a long list of link errors similar to the following:

```
../lib/libknitro.a(.text+0x28808): In function ‘ktr_xeb4’:
: undefined reference to ‘std::__default_alloc_template<true, 0>::deallo
cate(void*, unsigned int)’
../lib/libknitro.a(.text+0x28837): In function ‘ktr_xeb4’:
: undefined reference to ‘std::__default_alloc_template<true, 0>::deallo
cate(void*, unsigned int)’
../lib/libknitro.a(.text+0x290b0): more undefined references to ‘std::__
default_alloc_template<true, 0>::deallocate(void*, unsigned int)’ follow
../lib/libknitro.a(.text+0x2a0ff): In function ‘ktr_x1150’:
: undefined reference to ‘std::basic_string<char, std::char_traits<char>
```

```
, std::allocator<char> >::_S_empty_rep_storage'
../lib/libknitro.a(.text+0x2a283): In function `ktr_x1150':
: undefined reference to `std::__default_alloc_template<true, 0>::deallo
cate(void*, unsigned int)'
```

This indicates an incompatibility between the libstdc++ library on your Linux distribution and the library that KNITRO was built with. The incompatibilities may be caused by name-mangling differences between versions of the gcc compiler, and by differences in the Application Binary Interface of the two Linux distributions. The best fix is for Ziena to rebuild the KNITRO binaries on the same Linux distribution of your target machine (matching the Linux brand and release, and the gcc/g++ compiler versions). If you see these errors, please contact Ziena at info@ziena.com to correct the problem.

Another Linux link error sometimes seen when using the programs in examples/C is the following:

```
./callback1_dynamic: error while loading shared libraries: ../../lib/libmkl.so:
cannot restore segment prot after reloc: Permission denied
```

This is a security enhanced Linux (SELinux) error message. The Intel Math Kernel Library lib/libmkl.so shipped with KNITRO does not have the proper security identifiers for your distribution of SELinux (the library is loaded with user option `blasoption`). You could disable security enhancements, but a better fix is to change the security identifiers of the library to acceptable values. On Linux Fedora Core 4, an acceptable security type is "shlib_t"; other Linux distributions are probably similar. The fix is made by changing to the KNITRO lib directory and typing:

```
chcon -c -v -t shlib_t libmkl.so
```

2.4 Troubleshooting

Below are a list of steps to take if you are have difficulties installing KNITRO.

- Create an environment variable `ZIENA_LICENSE_DEBUG` and set it to 1. This will enable some debug output printing that will indicate where the license manager is looking for a license file. See Section 4.1 of the *Ziena License Manager User's Manual* for more details on how to set the `ZIENA_LICENSE_DEBUG` environment variable and generate debugging information.
- Ensure that the user has read access to all libraries and to the license file.
- Ensure that the program calling KNITRO is 32 (or 64) bit when KNITRO is 32 (or 64) bit.
- On Windows, please make sure that you are setting *system* environment variables rather than *user* environment variables, when setting environment variables for KNITRO.
- KNITRO has multiple options for installing license files. If the procedure you are trying is not working, please try an alternative procedure.
- If you have multiple KNITRO executable files or libraries of the same name on your computer, make sure that the one being used is really the one you intend to use (by making sure it appears first in the definition of the appropriate environment variable).

Please also refer to the *Ziena License Manager User's Manual* provided with your distribution for additional installation and troubleshooting information.

3 Using KNITRO with the AMPL modeling language

AMPL is a popular modeling language for optimization which allows users to represent their optimization problems in a user-friendly, readable, intuitive format. This makes the job of formulating and modeling a problem much simpler. For a description of AMPL see [7] or visit the AMPL web site at:

http://www.ampl.com/

It is straightforward to use KNITRO with the AMPL modeling language. We assume in the following that the user has successfully installed AMPL. The KNITRO/AMPL executable file `knitroampl` must be in the current directory where AMPL is started, or in a directory included in the `PATH` environment variable (such as a `bin` directory).

Inside of AMPL, to invoke the KNITRO solver type:

```
ampl:  option solver knitroampl;
```

at the prompt. To specify user options, type, for example,

```
ampl:  option knitro_options "maxit=100 alg=2";
```

The above command sets the maximum number of allowable iterations to 100 and chooses the Interior/CG algorithm (described in Section 8). When specifying multiple options, all options must be set with one `knitro_options` command as shown in the example above. If multiple `knitro_options` commands are specified in an AMPL session, only the last one will be read. See Section 3.2 for a summary of user specifiable options available in KNITRO for use with AMPL. For more detail on these options see Section 5. Note that in Section 5, user parameters are defined by text names such as `alg` and by programming language identifiers such as `KTR_PARAM_ALG`. In AMPL, parameters are set using only the (lowercase) text names, as specified in Section 3.2.

3.1 Example AMPL model solution using KNITRO

This section provides an example AMPL model and AMPL session which calls KNITRO to solve the problem:

$$\begin{aligned}
\underset{x}{\text{minimize}} \quad & 1000 - x_1^2 - 2x_2^2 - x_3^2 - x_1x_2 - x_1x_3 && \text{(3.2a)}\\
\text{subject to} \quad & 8x_1 + 14x_2 + 7x_3 - 56 = 0 && \text{(3.2b)}\\
& x_1^2 + x_2^2 + x_3^2 - 25 \geq 0 && \text{(3.2c)}\\
& x_1, x_2, x_3 \geq 0 && \text{(3.2d)}
\end{aligned}$$

with initial point $x = [x_1, x_2, x_3] = [2, 2, 2]$.

The AMPL model for the above problem is provided with KNITRO in a file called testproblem.mod, which is shown below.

AMPL test program file testproblem.mod

```
#
# Example problem formulated as an AMPL model used
# to demonstate using KNITRO with AMPL.
# The problem has two local solutions:
#   the point (0,0,8) with objective 936.0, and
#   the point (7,0,0) with objective 951.0

# Define variables and enforce that they be non-negative.
var x{j in 1..3} >= 0;

# Objective function to be minimized.
minimize obj:
     1000 - x[1]^2 - 2*x[2]^2 - x[3]^2 - x[1]*x[2] - x[1]*x[3];

# Equality constraint.
s.t. c1: 8*x[1] + 14*x[2] + 7*x[3] - 56 = 0;

# Inequality constraint.
s.t. c2: x[1]^2 + x[2]^2 + x[3]^2 -25 >= 0;

data;

# Define initial point.
let x[1] := 2;
let x[2] := 2;
let x[3] := 2;
```

The above example displays the ease with which an optimization problem can be expressed in the AMPL modeling language. Below is the AMPL session used to solve this problem with KNITRO. In the example below we set `alg=2` (to use the Interior/CG algorithm), `maxcrossit=2` (to refine the solution using the Active Set algorithm), and `outlev=1` (to limit output from KNITRO). See Section 7 for an explanation of the KNITRO output.

AMPL Example

```
ampl: reset;
ampl: option solver knitroampl;
ampl: option knitro_options "alg=2 maxcrossit=2 outlev=1";
ampl: model testproblem.mod;
ampl: solve;

KNITRO 7.0: alg=2
maxcrossit=2
outlev=1
```

```
=====================================
       Commercial Ziena License
            KNITRO 7.0.0
       Ziena Optimization, Inc.
=====================================

algorithm:    2
maxcrossit:   2
outlev:       1
KNITRO changing bar_murule from AUTO to 1.
KNITRO changing bar_initpt from AUTO to 2.

Problem Characteristics
-----------------------
Number of variables:                     3
    bounded below:                       3
    bounded above:                       0
    bounded below and above:             0
    fixed:                               0
    free:                                0
Number of constraints:                   2
    linear equalities:                   1
    nonlinear equalities:                0
    linear inequalities:                 0
    nonlinear inequalities:              1
    range:                               0
Number of nonzeros in Jacobian:          6
Number of nonzeros in Hessian:           5

EXIT: Locally optimal solution found.

Final Statistics
----------------
Final objective value               =   9.36000000000000e+02
Final feasibility error (abs / rel) =   0.00e+00 / 0.00e+00
Final optimality error  (abs / rel) =   3.55e-15 / 2.22e-16
# of iterations                     =          7
# of function evaluations           =          8
# of gradient evaluations           =          8
# of Hessian evaluations            =          7
Total program time (secs)           =       0.00321 (     0.001 CPU time)
Time spent in evaluations (secs)    =       0.00015

===============================================================================
```

```
KNITRO 7.0: Locally optimal solution found.
objective 9.360000e+02; feasibility error 0.000000e+00
7 major iterations; 8 function evaluations
ampl:
```

For descriptions of the KNITRO output see Section 7. To display the final solution variables `x` and the objective value `obj` through AMPL, use the AMPL `display` command as follows.

```
ampl: display x;
x [*] :=
1  0
2  0
3  8
;

ampl: display obj;
obj = 936
```

Upon completion, KNITRO displays a message and returns an exit code to AMPL. In the example above KNITRO found a solution, so the message was "`Locally optimal solution found`" with exit code of zero (exit code can be seen by typing "`ampl: display solve_result_num;`"). If a solution is not found, then KNITRO returns one of the following:

```
  0: Locally optimal solution found.
100: Current solution estimate cannot be improved.  Nearly optimal.
101: Relative change in feasible solution estimate < xtol.
102: Current feasible solution estimate cannot be improved.
200: Convergence to an infeasible point.  Problem may be locally infeasible.
201: Relative change in infeasible solution estimate < xtol.
202: Current infeasible solution estimate cannot be improved.
203: Multistart:  No primal feasible point found.
300: Problem appears to be unbounded.
400: Iteration limit reached.
401: Time limit reached.
403: MIP: All nodes have been explored.
404: MIP: Integer feasible point found.
405: MIP: Subproblem solve limit reached.
406: MIP: Node limit reached.
501: LP solver error.
502: Evaluation error.
503: Not enough memory.
504: Terminated by user.
505: Input or other API error.
506: Internal KNITRO error.
507: Unknown termination.
508: Illegal objno value.
```

Suppressing output when using KNITRO with AMPL

Even when setting the options:

```
ampl: option solver_msg 0;
ampl: option knitro_options "outlev=0";
```

in an AMPL session, AMPL will still print some basic information like the solver name and non-default user option settings to the screen. In order to suppress all AMPL and KNITRO output you must change your AMPL `solve` commands to something like

```
ampl: solve >scratch-file;
```

where `scratch-file` is the name of some temporary file where the unwanted output can be sent. Under Unix, `solve >/dev/null` automatically throws away the unwanted output, but under Windows you need to redirect output to an actual file.

3.2 KNITRO options for AMPL

KNITRO user options can be set from AMPL by typing the name of the option and a numeric value. When using AMPL's interactive mode, set all options in a single command; for example,

```
ampl:  option knitro_options "maxit=100 opttol=1.0e-5";
```

When running KNITRO directly with an AMPL problem, set user options on the command line with the problem name; for example,

```
knitroampl testproblem.nl maxit=100 opttol=1.0e-5
```

A complete list of available KNITRO options can always be shown by typing:

```
knitroampl -=
```

These options are summarized below in alphabetical order. All options specific to the barrier/interior-point algorithms start with `"bar_"`; options specific to mixed integer programming begin with `"mip_"`; and options specific to the multi-start procedure begin with `"ms_"`.

OPTION	DESCRIPTION	DEFAULT
alg algorithm	optimization algorithm used: 0: let KNITRO choose the algorithm 1: Interior/Direct (barrier) algorithm 2: Interior/CG (barrier) algorithm 3: Active Set algorithm	0
bar_directinterval	frequency for trying to force direct steps	10
bar_feasible	whether feasibility is given special emphasis: 0: no special emphasis on feasibility 1: iterates must honor inequalities 2: emphasize first getting feasible before optimizing 3: implement both options 1 and 2 above	0
bar_feasmodetol	tolerance for entering stay feasible mode	1.0e-4
bar_initmu	initial value for barrier parameter	1.0e-1
bar_initpt	initial point strategy for barrier algorithms 0: let KNITRO choose the initial point strategy 1: shift the initial point to improve barrier performance 2: do not alter the initial point supplied by the user	0
bar_maxbacktrack	maximum number of linesearch backtracks	3
bar_maxrefactor	maximum number of KKT refactorizations allowed	0
bar_murule	barrier parameter update rule: 0: let KNITRO choose the barrier update rule 1: monotone decrease rule 2: adaptive rule based on complementarity gap 3: probing rule (Interior/Direct only) 4: safeguarded Mehrotra predictor-corrector type rule 5: Mehrotra predictor-corrector type rule 6: rule based on minimizing a quality function	0

OPTION	DESCRIPTION	DEFAULT
bar_penaltycons	technique for penalizing constraints in the barrier algorithms: 0: let KNITRO choose the strategy 1: do not apply penalty approach to any constraints 2: apply a penalty approach to all general constraints	0
bar_penaltyrule	penalty parameter rule for step acceptance: 0: let KNITRO choose the strategy 1: use single penalty parameter approach 2: use more tolerant, flexible strategy	0
blasoption	specify the BLAS/LAPACK function library to use: 0: use KNITRO built-in functions 1: use Intel Math Kernel Library functions 2: use the dynamic library specified with "blasoptionlib"	1
debug	enable debugging output: 0: no extra debugging 1: print info to debug solution of the problem 2: print info to debug execution of the solver	0
delta	initial trust region radius scaling	1.0e0
feastol	feasibility termination tolerance (relative)	1.0e-6
feastol_abs	feasibility termination tolerance (absolute)	0.0e-0
gradopt	gradient computation method: 1: use exact gradients 2: compute forward finite-difference approximations 3: compute centered finite-difference approximations	1
hessopt	Hessian (Hessian-vector) computation method: 1: use exact Hessian derivatives 2: use dense quasi-Newton BFGS Hessian approximation 3: use dense quasi-Newton SR1 Hessian approximation 4: compute Hessian-vector products by finite diffs 5: compute exact Hessian-vector products 6: use limited-memory BFGS Hessian approximation	1
honorbnds	0: allow bounds to be violated during the optimization 1: enforce bounds satisfaction of all iterates 2: enforce bounds satisfaction of initial point	2
infeastol	tolerance for declaring infeasibility	1.0e-8
linsolver	linear system solver to use inside KNITRO: 0: let KNITRO choose the linear system solver 1: (not currently used; same as 0) 2: use a hybrid approach; solver depends on system 3: use a dense QR method (small problems only) 4: use HSL MA27 sparse symmetric indefinite solver 5: use HSL MA57 sparse symmetric indefinite solver	0
lmsize	number of limited-memory pairs stored in LBFGS approach	10
lpsolver	1: use internal LP solver in Active Set algorithm 2: use ILOG-CPLEX LP solver in Active Set algorithm (requires a valid CPLEX license) (specify library location with "cplexlibname")	1

OPTION	DESCRIPTION	DEFAULT
maxcgit	maximum allowable conjugate gradient (CG) iterations: 0: let KNITRO set the number based on the problem size n: maximum of $n > 0$ CG iterations per minor iteration	0
maxcrossit	maximum number of allowable crossover iterations	0
maxit	maximum number of iterations before terminating 0: let KNITRO set the number based on the problem n: maximum limit of $n > 0$ iterations	0
maxtime_cpu	maximum CPU time in seconds before terminating	1.0e8
maxtime_real	maximum real time in seconds before terminating	1.0e8
mip_branchrule	MIP branching rule: 0: let KNITRO choose the branching rule 1: most-fractional branching 2: pseudo-cost branching 3: strong branching	0
mip_debug	MIP debugging level 0: no MIP debugging output 1: print MIP debugging information	0
mip_gub_branch	Branch on GUBs 0: do not branch on GUB constraints 1: allow branching on GUB constraints	0
mip_heuristic	heuristic search approach 0: let KNITRO decide whether to apply a heuristic 1: do not apply any heuristic 2: use feasibility pump heuristic 3: use MPEC heuristic	0
mip_heuristic_maxit	heuristic search iteration limit	100
mip_implications	Add logical implications 0: do not add constraints from logical implications 1: add constraints from logical implications	1
mip_integer_tol	threshold for deciding integrality	1.0e-8
mip_integral_gap_abs	absolute integrality gap stop tolerance	1.0e-6
mip_integral_gap_rel	relative integrality gap stop tolerance	1.0e-6
mip_knapsack	add knapsack cuts 0: do not add knapsack cuts 1: add knapsack inequality cuts only 2: add knapsack inequality and equality cuts	1
mip_lpalg	LP subproblem algorithm 0: let KNITRO decide the LP algorithm 1: Interior/Direct (barrier) algorithm 2: Interior/CG (barrier) algorithm 3: Active Set (simplex) algorithm	0
mip_maxnodes	maximum nodes explored	100000
mip_maxsolves	maximum subproblem solves	200000
mip_maxtime_cpu	maximum CPU time in seconds for MIP	1.0e8
mip_maxtime_real	maximum real time in seconds for MIP	1.0e8

OPTION	DESCRIPTION	DEFAULT
mip_method	MIP method 0: let KNITRO choose the method 1: branch and bound method 2: hybrid method for convex nonlinear models	0
mip_outinterval	MIP node output interval	10
mip_outlevel	MIP output level	1
mip_outsub	enable MIP subproblem debug output	0
mip_pseudoinit	method to initialize pseudo-costs 0: let KNITRO choose the method 1: use average value 2: use strong branching	0
mip_rootalg	root node relaxation algorithm 0: let KNITRO decide the root algorithm 1: Interior/Direct (barrier) algorithm 2: Interior/CG (barrier) algorithm 3: Active Set algorithm	0
mip_rounding	MIP rounding rule 0: let KNITRO choose the rounding rule 1: do not attempt rounding 2: use fast heuristic 3: apply rounding solve selectively 4: apply rounding solve always	0
mip_selectrule	MIP node selection rule 0: let KNITRO choose the node select rule 1: use depth first search 2: use best bound node selection 3: use a combination of depth first and best bound	0
mip_strong_candlim	strong branching candidate limit	10
mip_strong_level	strong branching level limit	10
mip_strong_maxit	strong branching subproblem iteration limit	1000
mip_terminate	termination condition for MIP 0: terminate at optimal solution 1: terminate at first integer feasible solution	0
ms_enable	0: multi-start not enabled 1: multi-start enabled	0
ms_maxbndrange	maximum range to vary unbounded x when generating start points	1.0e3
ms_maxsolves	maximum number of start points to try during multi-start 0: let KNITRO set the number based on problem size n: try exactly $n > 0$ start points	0
ms_maxtime_cpu	maximum CPU time for multi-start, in seconds	1.0e8
ms_maxtime_real	maximum real time for multi-start, in seconds	1.0e8
ms_num_to_save	number feasible points to save in "knitro_mspoints.log"	0
ms_savetol	tolerance for feasible points to be considered distinct	1.0e-6
ms_startptrange	maximum range to vary all x when generating start points	1.0e20
ms_terminate	termination condition for multi-start 0: terminate after ms_maxsolves 1: terminate at first local optimum (if before ms_maxsolves) 2: terminate at first feasible solution (if before ms_maxsolves)	0

OPTION	DESCRIPTION	DEFAULT
newpoint	0: no action 1: save the latest new point to file "knitro_newpoint.log" 2: append all new points to file "knitro_newpoint.log"	0
objrange	maximum allowable objective function magnitude	1.0e20
opttol	optimality termination tolerance (relative)	1.0e-6
opttol_abs	optimality termination tolerance (absolute)	0.0e-0
outappend	append output to existing files: 0: do not append 1: do append	0
outdir	directory where output files are created	
outlev	printing output level: 0: no printing 1: just print summary information 2: print basic information every 10 iterations 3: print basic information at each iteration 4: print all information at each iteration 5: also print final (primal) variables 6: also print final Lagrange multipliers (sensitivies)	2
outmode	0: direct KNITRO output to standard out (e.g., screen) 1: direct KNITRO output to the file "knitro.log" 2: print to both the screen and file "knitro.log"	0
pivot	initial pivot threshold for matrix factorizations	1.0e-8
presolve_dbg	0: no debugging information 2: print the KNITRO problem with AMPL model names	0
scale	0: do not scale the problem 1: perform automatic scaling of functions	1
soc	0: do not allow second order correction steps 1: selectively try second order correction steps 2: always try second order correction steps	1
xtol	stepsize termination tolerance	1.0e-15

3.3 Solving with complementarity constraints

KNITRO is able to solve mathematical programs with complementarity constraints (MPCCs) through the AMPL interface. A complementarity constraint enforces that two variables are *complementary* to each other; i.e., that the following conditions hold for scalar variables x and y:

$$x \times y = 0, \quad x \geq 0, \quad y \geq 0. \tag{3.3}$$

The condition above is sometimes expressed more compactly as

$$0 \leq x \quad \perp \quad y \geq 0.$$

See Section 10.5 for more information about the mathematics of complementarity constraints.

These constraints must be formulated in a particular way through AMPL in order for KNITRO to effectively deal with them. In particular, complementarity constraints should be modeled using the AMPL `complements` command; e.g.,

```
0 <= x complements y >= 0;
```

and they *must* be formulated as one variable complementary to another variable. They *may not* be formulated as a function complementary to a variable or a function complementary to a function. KNITRO will print a warning if functions are used in complementarity constraints, but it is not able to fix the problem. If a complementarity involves a function $F(x)$, for example,

$$0 \leq F(x) \quad \perp \quad x \geq 0,$$

then the user should reformulate the AMPL model by adding a slack variable, as shown below, so that it is formulated as a variable complementary to another variable:

```
var x; var s;
...
constraint_name_a: F(x) = s;
constraint_name_b: 0 <= s complements x >= 0;
```

Be aware that the AMPL presolver sometimes removes complementarity constraints. Check carefully that the problem definition reported by KNITRO includes all complementarity constraints, or switch off the AMPL presolver to be safe ("`option presolve 0;`").

3.4 Specifying MIP branching priorities

It is possible to specify branching priorities for integer variables when solving a mixed integer program (MIP) in KNITRO by using the AMPL suffixes feature. Assume we have a MIP model formulated in AMPL with three integer variables, (y_1, y_2, y_3). We can use the AMPL `.priority` suffix feature to specify branching priority values for these integer variables as shown below after reading in the AMPL model.

```
...
ampl: suffix priority IN, integer, >=0, <=9999;
ampl: let y[1].priority := 5;
ampl: let y[2].priority := 1;
ampl: let y[3].priority := 10;
```

Priorities must be positive numbers (variables with non-positive values are ignored). Variables with higher priority values will be considered for branching before variables with lower priority values. When priorities for a subset of variables are equal, the branching rule is applied as a tiebreaker. In the example below, y_3 is given the highest branching priority and y_2 is given the lowest branching priority. See the AMPL documentation for more information on the `.priority` suffix.

3.5 Displaying AMPL variables in KNITRO

AMPL will often perform a reordering of the variables and constraints defined in the AMPL model. The AMPL presolver may also simplify the form of the problem by eliminating certain variables or constraints. The output printed by KNITRO corresponds to the reordered, reformulated problem. To view final variable and constraint values in the original AMPL model, use the AMPL `display` command after KNITRO has completed solving the problem.

It is possible to correlate KNITRO variables and constraints with the original AMPL model. You must type an extra command in the AMPL session ("`option knitroampl_auxfiles rc;`"), and set KNITRO option `presolve_dbg=2`. Then the solver will print the variables and constraints that KNITRO receives, with their upper and lower bounds, and their AMPL model names. The extra AMPL command causes the model names to be passed to the KNITRO/AMPL solver.

The output below is obtained with the example file testproblem.mod supplied with your distribution. The center column of variable and constraint names are those used by KNITRO, while the names in the right-hand column are from the AMPL model:

```
ampl: model testproblem.mod;
ampl: option solver knitroampl;
ampl: option knitroampl_auxfiles rc;
ampl: option knitro_options "presolve_dbg=2 outlev=0";

KNITRO 7.0: presolve_dbg=2
outlev=0
----- AMPL problem for KNITRO -----
Objective name:  obj
   0.000000e+00  <=  x[   0]  <=     1.000000e+20  x[1]
   0.000000e+00  <=  x[   1]  <=     1.000000e+20  x[2]
   0.000000e+00  <=  x[   2]  <=     1.000000e+20  x[3]

   2.500000e+01  <=  c[   0]  <=     1.000000e+20  c2
   5.600000e+01  <=  c[   1]  <=     5.600000e+01  c1
-----------------------------------
KNITRO 7.0: Locally optimal solution found.
objective 9.360000e+02; feasibility error 7.105427e-15
6 major iterations; 7 function evaluations
```

4 The KNITRO callable library

This section includes information on how to embed and call the KNITRO solver from inside a program. KNITRO is written in C and C++, with a well-documented application programming interface (API) defined in the file knitro.h. The KNITRO product contains example interfaces written in various programming languages under the directory /examples. These are briefly discussed in the following sections (C in 4.2 and 4.3, C++ in 4.4, Java in 4.5, and Fortran in 4.6). Each example consists of a main driver program coded in the given language that defines an optimization problem and invokes KNITRO to solve it. Examples also contain a makefile illustrating how to link the KNITRO library with the target language driver program.

In all languages KNITRO runs as a thread-safe module, which means that the calling program can create multiple instances of a KNITRO solver in different threads, each instance solving a different problem. This is useful in a multiprocessing environment; for instance, in a web application server.

4.1 KNITRO in a C application

The KNITRO callable library is typically used to solve an optimization problem through a sequence of four basic function calls:

- `KTR_new()`: create a new KNITRO solver context pointer, allocating resources
- `KTR_init_problem()` or `KTR_mip_init_problem()`: load the problem definition into the KNITRO solver
- `KTR_solve()` or `KTR_mip_solve`: solve the problem
- `KTR_free()`: delete the KNITRO context pointer, releasing allocated resources

The functions `KTR_init_problem()` and `KTR_solve()` are used for continuous optimization models, while `KTR_mip_init_problem()` and `KTR_mip_solve()` are for optimization models with one or more integer variables. The complete C language API is defined in the file knitro.h, provided in the installation under the /include directory. Functions for setting and getting user options are described in Sections 5.2 and 5.3. Functions for retrieving KNITRO results are described in Section 7.3 and illustrated in the examples/C files. The remainder of this section describes in detail the four basic function calls.

```
KTR_context_ptr KTR_new (void)
```

This function must be called first. It returns a pointer to an object (the KNITRO "context pointer") that is used in all other calls. If you enable KNITRO with the Ziena floating network license handler, then this call also checks out a license and reserves it until `KTR_free()` is called with the context pointer, or the program ends. The contents of the context pointer should never be modified by a calling program.

```
int KTR_free (KTR_context_ptr * kc_handle)
```

This function should be called last and will free the context pointer. The address of the context pointer is passed so that KNITRO can set it to NULL after freeing all memory. This prevents the application from mistakenly calling KNITRO functions after the context pointer has been freed.

The C interface for KNITRO requires the application to define an optimization problem (1.1) in the following general format (for complementarity constraints, see Section 10.5):

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) && \text{(4.4a)} \\ \text{subject to} \quad & \texttt{cLoBnds} \le c(x) \le \texttt{cUpBnds} && \text{(4.4b)} \\ & \texttt{xLoBnds} \le x \le \texttt{xUpBnds} && \text{(4.4c)} \end{aligned}$$

where `cLoBnds` and `cUpBnds` are vectors of length m, and `xLoBnds` and `xUpBnds` are vectors of length n. If constraint i is an equality constraint, set $\texttt{cLoBnds}[i] = \texttt{cUpBnds}[i]$. If constraint i is unbounded from below or above, set $\texttt{cLoBnds}[i]$ or $\texttt{cUpBnds}[i]$ to the value `-KTR_INFBOUND` or `KTR_INFBOUND`, respectively. Similarly, for `xLoBnds` and `xUpBnds`. The constant `KTR_INFBOUND` is defined in `knitro.h` and stands for infinity in the KNITRO code.

To use KNITRO the application must provide routines for evaluating the objective $f(x)$ and constraint functions $c(x)$. For best performance, the application should also provide routines to evaluate first derivatives (gradients of $f(x)$ and $c(x)$), and, ideally, the second derivatives (Hessian of the Lagrangian). First derivatives in the C language API are denoted by `objGrad` and `jac`, where $\texttt{objGrad} = \nabla f(x)$, and `jac` is the $m \times n$ Jacobian matrix of constraint gradients such that the i-th row equals $\nabla c_i(x)$.

The Hessian of the Lagrangian is a matrix constructed from the individual second derivative matrices of the objective and constraint functions

$$H(x, \lambda) = \nabla^2 f(x) + \sum_{i=0}^{m-1} \lambda_i \nabla^2 c_i(x), \tag{4.5}$$

and is denoted by `hess` in the C language API. Here λ is the vector of Lagrange multipliers (dual variables). See Section 4.8 for further details on constructing the Jacobian and Hessian matrix in sparse form.

The ability to provide exact first derivatives is essential for efficient and reliable performance. Packages like ADOL-C and ADIFOR can help in generating code with derivatives. If the user is unable or unwilling to provide exact first derivatives, KNITRO provides an option that computes approximate first derivatives using finite-differencing (see Sections 4.10 and 9.1).

Exact second derivatives are less important, as KNITRO provides several options that substitute quasi-Newton approximations for the Hessian (see Section 9.2). However, the ability to provide exact second derivatives often dramatically improves the performance of KNITRO.

Functions KTR_init_problem and KTR_mip_init_problem:

```
int KTR_init_problem
      ( KTR_context_ptr kc,
        int n,
        int objGoal,
        int objType,
        double * xLoBnds,
        double * xUpBnds,
        int m,
        int * cType,
        double * cLoBnds,
        double * cUpBnds,
        int nnzJ,
        int * jacIndexVars,
        int * jacIndexCons,
        int nnzH,
        int * hessIndexRows,
        int * hessIndexCols,
        double * xInitial,
        double * lambdaInitial )
```

```
int KTR_mip_init_problem
      ( KTR_context_ptr kc,
        int n,
        int objGoal,
        int objType,
        int objFnType,
        int * xType,
        double * xLoBnds,
        double * xUpBnds,
        int m,
        int * cType,
        int * cFnType,
        double * cLoBnds,
        double * cUpBnds,
        int nnzJ,
        int * jacIndexVars,
        int * jacIndexCons,
        int nnzH,
        int * hessIndexRows,
        int * hessIndexCols,
        double * xInitial,
        double * lambdaInitial )
```

These functions pass the optimization problem definition to KNITRO, where it is copied and stored internally until `KTR_free()` is called. Once initialized, the problem may be solved any number of times with different user options or initial points (see the `KTR_restart()` call below). Array arguments passed to `KTR_init_problem()` or `KTR_mip_init_problem()` are not referenced again and may be freed or reused if desired. In the description below, some programming macros are mentioned as alternatives to fixed numeric constants; e.g., `KTR_OBJGOAL_MINIMIZE`. These macros are defined in knitro.h

Arguments:

`KTR_context_ptr kc:` is the KNITRO context pointer. Do not modify its contents.

`int n:` is a scalar specifying the number of variables in the problem; i.e., the length of x in (4.4).

`int objGoal:` is the optimization goal.

- `0:` if the goal is to minimize the objective function (`KTR_OBJGOAL_MINIMIZE`)
- `1:` if the goal is to maximize the objective function (`KTR_OBJGOAL_MAXIMIZE`)

`int objType:` is a scalar that describes the type of objective function $f(x)$ in (4.4).

- `0:` if $f(x)$ is a nonlinear function or its type is unknown (`KTR_OBJTYPE_GENERAL`)
- `1:` if $f(x)$ is a linear function (`KTR_OBJTYPE_LINEAR`)

2: if $f(x)$ is a quadratic function (KTR_OBJTYPE_QUADRATIC)

int objFnType: is a scalar that describes the convexity status of the objective function $f(x)$ in (4.4) (**MIP only**). See Section 4.9.

0: if the convexity status of $f(x)$ is unknown (KTR_FNTYPE_UNCERTAIN)

1: if $f(x)$ is a convex function when minimizing (KTR_FNTYPE_CONVEX)

2: if $f(x)$ is not a convex function when minimizing (KTR_FNTYPE_NONCONVEX)

int * xType: is an array of length n that describes the types of variables x in (4.4) (**MIP only**).

0: if x_i is a continuous variable (KTR_VARTYPE_CONTINUOUS)

1: if x_i is an integer variable (KTR_VARTYPE_INTEGER)

2: if x_i is a binary variable (KTR_VARTYPE_BINARY)

double * xLoBnds: is an array of length n specifying the lower bounds on x. xLoBnds[i] must be set to the lower bound of the corresponding i-th variable x_i. If the variable has no lower bound, set xLoBnds[i] to be -KTR_INFBOUND (defined in knitro.h). For binary variables, set xLoBnds[i]=0.

double * xUpBnds: is an array of length n specifying the upper bounds on x. xUpBnds[i] must be set to the upper bound of the corresponding i-th variable x_i. If the variable has no upper bound, set xUpBnds[i] to be KTR_INFBOUND (defined in knitro.h). For binary variables, set xUpBnds[i]=1.

int m: is a scalar specifying the number of constraints $c(x)$ in (4.4).

int * cType: is an array of length m that describes the types of the constraint functions $c(x)$ in (4.4).

0: if $c_i(x)$ is a nonlinear function or its type is unknown (KTR_CONTYPE_GENERAL)

1: if $c_i(x)$ is a linear function (KTR_CONTYPE_LINEAR)

2: if $c_i(x)$ is a quadratic function (KTR_CONTYPE_QUADRATIC)

int * cFnType: is an array of length m that describes the convexity status of the constraint functions $c(x)$ in (4.4) (**MIP only**). See Section 4.9.

0: if the convexity of constraint $c_i^L \leq c_i(x) \leq c_i^U$ is unknown (KTR_FNTYPE_UNCERTAIN)

1: if constraint $c_i^L \leq c_i(x) \leq c_i^U$ is a convex constraint (KTR_FNTYPE_CONVEX)

2: if constraint $c_i^L \leq c_i(x) \leq c_i^U$ is not a convex constraint (KTR_FNTYPE_NONCONVEX)

double * cLoBnds: is an array of length m specifying the lower bounds on the constraints $c(x)$ in (4.4). cLoBnds[i] must be set to the lower bound of the corresponding i-th constraint. If the constraint has no lower bound, set cLoBnds[i] to be -KTR_INFBOUND (defined in knitro.h). If the constraint is an equality, then cLoBnds[i] should equal cUpBnds[i].

double * cUpBnds: is an array of length m specifying the upper bounds on the constraints $c(x)$ in (4.4). cUpBnds[i] must be set to the upper bound of the corresponding i-th constraint. If the constraint has no upper bound, set cUpBnds[i] to be KTR_INFBOUND (defined in knitro.h). If the constraint is an equality, then cLoBnds[i] should equal cUpBnds[i].

`int nnzJ`: is a scalar specifying the number of nonzero elements in the sparse constraint Jacobian. See Section 4.8.

`int * jacIndexVars`: is an array of length `nnzJ` specifying the variable indices of the constraint Jacobian nonzeroes. If `jacIndexVars`[i]=j, then `jac`[i] refers to the j-th variable, where `jac` is the array of constraint Jacobian nonzero elements passed in the call `KTR_solve()`.

`jacIndexCons`[i] and `jacIndexVars`[i] determine the row numbers and the column numbers, respectively, of the nonzero constraint Jacobian element `jac`[i]. See Section 4.8.

NOTE: C array numbering starts with index 0. Therefore, the j-th variable x_j maps to array element `x[j]`, and $0 \leq$ `j` $<$ `n`.

`int * jacIndexCons`: is an array of length `nnzJ` specifying the constraint indices of the constraint Jacobian nonzeroes. If `jacIndexCons`[i]=k, then `jac`[i] refers to the k-th constraint, where `jac` is the array of constraint Jacobian nonzero elements passed in the call `KTR_solve()`.

`jacIndexCons`[i] and `jacIndexVars`[i] determine the row numbers and the column numbers, respectively, of the nonzero constraint Jacobian element `jac`[i]. See Section 4.8.

NOTE: C array numbering starts with index 0. Therefore, the k-th constraint c_k maps to array element `c[k]`, and $0 \leq$ `k` $<$ `m`.

`int nnzH`: is a scalar specifying the number of nonzero elements in the sparse Hessian of the Lagrangian. Only nonzeroes in the upper triangle (including diagonal nonzeroes) should be counted. See Section 4.8.

NOTE: If user option `hessopt` is not set to `KTR_HESSOPT_EXACT`, then Hessian nonzeroes will not be used (see Section 5.1). In this case, set `nnzH=0`, and pass NULL pointers for `hessIndexRows` and `hessIndexCols`.

`int * hessIndexRows`: is an array of length `nnzH` specifying the row number indices of the Hessian nonzeroes.

`hessIndexRows`[i] and `hessIndexCols`[i] determine the row numbers and the column numbers, respectively, of the nonzero Hessian element `hess`[i], where `hess` is the array of Hessian elements passed in the call `KTR_solve()`. See Section 4.8.

NOTE: Row numbers are in the range 0 .. `n` $-$ 1.

`int * hessIndexCols`: is an array of length `nnzH` specifying the column number indices of the Hessian nonzeroes.

`hessIndexRows`[i] and `hessIndexCols`[i] determine the row numbers and the column numbers, respectively, of the nonzero Hessian element `hess`[i], where `hess` is the array of Hessian elements passed in the call `KTR_solve()`. See Section 4.8.

NOTE: Column numbers are in the range 0 .. `n` $-$ 1.

`double * xInitial`: is an array of length `n` containing an initial guess of the solution vector x. If the application prefers to let KNITRO make an initial guess, then pass a NULL pointer for `xInitial`.

`double * lambdaInitial:` is an array of length `m+n` containing an initial guess of the Lagrange multipliers for the constraints $c(x)$ (4.4*b*) and bounds on the variables x (4.4*c*). The first `m` components of `lambdaInitial` are multipliers corresponding to the constraints specified in $c(x)$, while the last `n` components are multipliers corresponding to the bounds on x. If the application prefers to let KNITRO make an initial guess, then pass a NULL pointer for `lambdaInitial`.

To solve the nonlinear optimization problem (4.4), KNITRO needs the application to supply information at various trial points. KNITRO specifies a trial point with a new vector of variable values x, and sometimes a corresponding vector of Lagrange multipliers λ. At a trial point, KNITRO may ask the application to:

`KTR_RC_EVALFC`:	Evaluate f and c at x.
`KTR_RC_EVALGA`:	Evaluate ∇f and ∇c at x.
`KTR_RC_EVALH`:	Evaluate the Hessian matrix of the problem at x and λ.
`KTR_RC_EVALHV`:	Evaluate the Hessian matrix times a vector v at x and λ.

The constants `KTR_RC_*` are return codes defined in knitro.h.

The KNITRO C language API has two modes of operation for obtaining problem information: "**callback**" and "**reverse communication**". With callback mode the application provides C language function pointers that KNITRO may call to evaluate the functions, gradients, and Hessians. With reverse communication, the function `KTR_solve()` (or `KTR_mip_solve()`) returns one of the constants listed above to tell the application what it needs, and then waits to be called again with the new problem information. For more details, see Section 9.8 (callback mode) and Section 9.7 (reverse communication mode). Both modes use `KTR_solve()` (or `KTR_mip_solve()` for MIP solves).

Functions KTR_solve and KTR_mip_solve:

Functions `KTR_solve()` and `KTR_mip_solve()` have the same parameter list (for convenience, we just show `KTR_solve()` below). `KTR_solve()` should be used for models where all the variables are continuous, while `KTR_mip_solve()` should be used for models with one or more binary or integer variables.

```
int KTR_solve(  KTR_context_ptr kc,     /*input*/
                double * x,             /*output*/
                double * lambda,        /*output*/
                int evalStatus,         /*input, reverse comm only*/
                double * obj,           /*input and output*/
                double * c,             /*input, reverse comm only*/
                double * objGrad,       /*input, reverse comm only*/
                double * jac,           /*input, reverse comm only*/
                double * hess,          /*input, reverse comm only*/
                double * hessVector,    /*input, output, rev comm*/
                void * userParams )     /*input, callback only*/
```

Arguments:

`KTR_context_ptr kc:` is the KNITRO context pointer. Do not modify its contents.

`double * x:` is an array of length `n` output by KNITRO. If `KTR_solve()` returns `KTR_RC_OPTIMAL` (zero), then `x` contains the solution.

Reverse communications mode: upon return, `x` contains the value of unknowns at which KNITRO needs more problem information. For continuous problems, if user option `newpoint` is set to `KTR_NEWPOINT_USER` (see Section 5.1) and `KTR_solve()` returns `KTR_RC_NEWPOINT`, then `x` contains a newly accepted iterate, but not the final solution.

`double * lambda:` is an array of length `m+n` output by KNITRO. If `KTR_solve()` returns zero, then `lambda` contains the multiplier values at the solution. The first `m` components of `lambda` are multipliers corresponding to the constraints specified in $c(x)$, while the last `n` components are multipliers corresponding to the bounds on x.

Reverse communications mode: upon return, `lambda` contains the value of multipliers at which KNITRO needs more problem information.

`int evalStatus:` is a scalar input to KNITRO used only in **reverse communications mode**. A value of zero means the application successfully computed the problem information requested by KNITRO at `x` and `lambda`. A nonzero value means the application failed to compute problem information (e.g., if a function is undefined at the requested value `x`).

`double * obj:` is a scalar holding the value of $f(x)$ at the current `x`. If `KTR_solve()` returns `KTR_RC_OPTIMAL` (zero), then `obj` contains the value of the objective function $f(x)$ at the solution.

Reverse communications mode: if `KTR_solve()` returns `KTR_RC_EVALFC`, then `obj` must be filled with the value of $f(x)$ computed at `x` before `KTR_solve()` is called again.

`double * c:` is an array of length `m` used only in **reverse communications mode**. If `KTR_solve()` returns `KTR_RC_EVALFC`, then `c` must be filled with the value of $c(x)$ computed at `x` before `KTR_solve()` is called again.

`double * objGrad:` is an array of length `n` used only in **reverse communications mode**. If `KTR_solve()` returns `KTR_RC_EVALGA`, then `objGrad` must be filled with the value of $\nabla f(x)$ computed at `x` before `KTR_solve()` is called again.

`double * jac:` is an array of length `nnzJ` used only in **reverse communications mode**. If `KTR_solve()` returns `KTR_RC_EVALGA`, then `jac` must be filled with the constraint Jacobian $\nabla c(x)$ computed at `x` before `KTR_solve()` is called again. Entries are stored according to the sparsity pattern defined in `KTR_init_problem()`.

`double * hess:` is an array of length `nnzH` used only in **reverse communications mode**, and only if option `hessopt` is set to `KTR_HESSOPT_EXACT` (see Section 5.1). If `KTR_solve()` returns `KTR_RC_EVALH`, then `hess` must be filled with the Hessian of the Lagrangian computed at `x` and `lambda` before `KTR_solve()` is called again. Entries are stored according to the sparsity pattern defined in `KTR_init_problem()`.

`double * hessVector:` is an array of length `n` used only in **reverse communications mode**, and only if option `hessopt` is set to `KTR_HESSOPT_PRODUCT` (see Section 5.1). If `KTR_solve()` returns `KTR_RC_EVALHV`, then the Hessian of the Lagrangian at `x` and `lambda` should be

multiplied by `hessVector`, and the result placed in `hessVector` before `KTR_solve()` is called again.

`void * userParams`: is a pointer to a structure used only in **callback mode**. The pointer is provided so the application can pass additional parameters needed for its callback routines. If the application needs no additional parameters, then pass a NULL pointer. See Section 9.8 for more details.

Return Value:

The return value of `KTR_solve()`/`KTR_mip_solve()` specifies the final exit code from the optimization process. If the return value is zero (`KTR_RC_OPTIMAL`) or negative, then KNITRO has finished solving. In **reverse communications mode** the return value may be positive, in which case it specifies a request for additional problem information, after which the application should call KNITRO again. A detailed description of the possible return values is given in the appendix.

Function KTR_restart:

```
int KTR_restart(  KTR_context_ptr kc,
                  double * x,
                  double * lambda )
```

This function can be called to start another `KTR_solve()`/`KTR_mip_solve()` sequence after making small modifications. The problem structure cannot be changed (e.g., `KTR_init_problem()`/`KTR_mip_init_problem()` cannot be called between `KTR_solve()`/`KTR_mip_solve()` and `KTR_restart()`). However, user options can be modified, and a new initial value can be passed with `KTR_restart()`. The sample program examples/C/restartExample.c uses `KTR_restart()` to solve the same problem from the same start point, but each time changing the interior point `bar_murule` option to a different value.

Function KTR_mip_set_branching_priorities:

```
int  KTR_mip_set_branching_priorities
    (      KTR_context_ptr kc,
     const int * const     xPriorities);
```

This function can be used to set the branching priorities for integer variables when using the MIP features in KNITRO. Priorities must be positive numbers (variables with non-positive values are ignored). Variables with higher priority values will be considered for branching before variables with lower priority values. When priorities for a subset of variables are equal, the branching rule is applied as a tiebreaker. Array `xPriorities` has length `n`, and values for continuous variables are ignored. KNITRO makes a local copy of all inputs, so the application may free memory after the call. This routine must be called after calling `KTR_mip_init_problem()` and before calling `KTR_mip_solve()`.

4.2 Example in C using callback mode

The KNITRO distribution comes with several C language programs in the directory examples/C. The instructions in examples/C/README.txt explain how to compile and run the examples. This section overviews the coding of driver programs using the callback interface, but the working examples provide more complete detail.

Consider the following nonlinear optimization problem from the Hock and Schittkowski test set [9]:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 100 - (x_2 - x_1^2)^2 + (1 - x_1)^2 && \text{(4.6a)} \\ \text{subject to} \quad & 1 \leq x_1 x_2 && \text{(4.6b)} \\ & 0 \leq x_1 + x_2^2 && \text{(4.6c)} \\ & x_1 \leq 0.5. && \text{(4.6d)} \end{aligned}$$

This problem is coded as examples/C/problemHS15.c.

Every driver starts by allocating a new KNITRO solver instance and checking that it succeeded (`KTR_new()` might return `NULL` if the Ziena license check fails):

```
#include "knitro.h"

/*... Include other headers, define main() ...*/

KTR_context  *kc;

/*... Declare other local variables ...*/

/*---- CREATE A NEW KNITRO SOLVER INSTANCE. */
kc = KTR_new();
if (kc == NULL)
    {
    printf ("Failed to find a Ziena license.\n");
    return( -1 );
    }
```

The next task is to load the problem definition into the solver using `KTR_init_problem()`. The problem has 2 unknowns and 2 constraints, and it is easily seen that all first and second partial derivatives are generally nonzero. The code below captures the problem definition and passes it to KNITRO:

```
/*---- DEFINE PROBLEM SIZES. */
n = 2;
m = 2;
nnzJ = 4;
nnzH = 3;

/*... allocate memory for xLoBnds, xUpBnds, etc. ...*/

/*---- DEFINE THE OBJECTIVE FUNCTION AND VARIABLE BOUNDS. */
objType = KTR_OBJTYPE_GENERAL;
objGoal = KTR_OBJGOAL_MINIMIZE;
xLoBnds[0] = -KTR_INFBOUND;
xLoBnds[1] = -KTR_INFBOUND;
```

```
xUpBnds[0] = 0.5;
xUpBnds[1] = KTR_INFBOUND;

/*---- DEFINE THE CONSTRAINT FUNCTIONS. */
cType[0] = KTR_CONTYPE_QUADRATIC;
cType[1] = KTR_CONTYPE_QUADRATIC;
cLoBnds[0] = 1.0;
cLoBnds[1] = 0.0;
cUpBnds[0] = KTR_INFBOUND;
cUpBnds[1] = KTR_INFBOUND;

/*---- PROVIDE FIRST DERIVATIVE STRUCTURAL INFORMATION. */
jacIndexCons[0] = 0;
jacIndexCons[1] = 0;
jacIndexCons[2] = 1;
jacIndexCons[3] = 1;
jacIndexVars[0] = 0;
jacIndexVars[1] = 1;
jacIndexVars[2] = 0;
jacIndexVars[3] = 1;

/*---- PROVIDE SECOND DERIVATIVE STRUCTURAL INFORMATION. */
hessIndexRows[0] = 0;
hessIndexRows[1] = 0;
hessIndexRows[2] = 1;
hessIndexCols[0] = 0;
hessIndexCols[1] = 1;
hessIndexCols[2] = 1;

/*---- CHOOSE AN INITIAL START POINT. */
xInitial[0] = -2.0;
xInitial[1] =  1.0;

/*---- INITIALIZE KNITRO WITH THE PROBLEM DEFINITION. */
nStatus = KTR_init_problem (kc, n, objGoal, objType,
                            xLoBnds, xUpBnds,
                            m, cType, cLoBnds, cUpBnds,
                            nnzJ, jacIndexVars, jacIndexCons,
                            nnzH, hessIndexRows, hessIndexCols,
                            xInitial, NULL);
if (nStatus != 0)
    { /*... an error occurred ...*/ }

/*... free xLoBnds, xUpBnds, etc. ...*/
```

Assume for simplicity that the user writes three routines for computing problem information. In examples/C/problemHS15.c these are named `computeFC`, `computeGA`, and `computeH`. To write

a driver program using **callback** mode, simply wrap each evaluation routine in a function that matches the KTR_callback() prototype defined in knitro.h. Note that all three wrappers use the same prototype. This is in case the application finds it convenient to combine some of the evaluation steps, as demonstrated in examples/C/callbackExample2.c.

```
/*-----------------------------------------------------------------*/
/*     FUNCTION callbackEvalFC                                     */
/*-----------------------------------------------------------------*/
/** The signature of this function matches KTR_callback in knitro.h.
 *  Only "obj" and "c" are modified.
 */
int  callbackEvalFC (const int             evalRequestCode,
                     const int             n,
                     const int             m,
                     const int             nnzJ,
                     const int             nnzH,
                     const double * const  x,
                     const double * const  lambda,
                           double * const  obj,
                           double * const  c,
                           double * const  objGrad,
                           double * const  jac,
                           double * const  hessian,
                           double * const  hessVector,
                           void   *        userParams)
{
    if (evalRequestCode != KTR_RC_EVALFC)
      {
      printf ("*** callbackEvalFC incorrectly called with eval code %d\n",
              evalRequestCode);
      return( -1 );
      }

    /*---- IN THIS EXAMPLE, CALL THE ROUTINE IN problemDef.h. */
    *obj = computeFC (x, c);
    return( 0 );
}


/*-----------------------------------------------------------------*/
/*     FUNCTION callbackEvalGA                                     */
/*-----------------------------------------------------------------*/
/** The signature of this function matches KTR_callback in knitro.h.
 *  Only "objGrad" and "jac" are modified.
 */

    /*... similar implementation to callbackEvalFC ...*/
```

```
/*-------------------------------------------------------------------*/
/*     FUNCTION callbackEvalH                                        */
/*-------------------------------------------------------------------*/
/** The signature of this function matches KTR_callback in knitro.h.
 *  Only "hessian" is modified.
 */

    /*... similar implementation to callbackEvalFC ...*/
```

Back in the main program each wrapper function is registered as a callback to KNITRO, and then `KTR_solve()` is invoked to find the solution:

```
    /*---- REGISTER THE CALLBACK FUNCTIONS THAT PERFORM PROBLEM EVALS.
     *---- THE HESSIAN CALLBACK ONLY NEEDS TO BE REGISTERED FOR SPECIFIC
     *---- HESSIAN OPTIONS (E.G., IT IS NOT REGISTERED IF THE OPTION FOR
     *---- BFGS HESSIAN APPROXIMATIONS IS SELECTED).
     */
    if (KTR_set_func_callback (kc, &callbackEvalFC) != 0)
        exit( -1 );
    if (KTR_set_grad_callback (kc, &callbackEvalGA) != 0)
        exit( -1 );
    if ((nHessOpt == KTR_HESSOPT_EXACT) ||
        (nHessOpt == KTR_HESSOPT_PRODUCT))
        {
        if (KTR_set_hess_callback (kc, &callbackEvalHess) != 0)
            exit( -1 );
        }

    /*---- SOLVE THE PROBLEM.
     */
    nStatus = KTR_solve (kc, x, lambda, 0, &obj,
                         NULL, NULL, NULL, NULL, NULL, NULL);
    if (nStatus != KTR_RC_OPTIMAL)
      printf ("KNITRO failed to solve the problem, final status = %d\n",
              nStatus);

    /*---- DELETE THE KNITRO SOLVER INSTANCE. */
    KTR_free (&kc);
```

This completes the brief overview of creating driver programs to run KNITRO in C using callback mode. Again, more details and options are demonstrated in the programs located in examples/C, including an example for a mixed integer nonlinear programming model. Outputs produced when running KNITRO are discussed in Section 7.

4.3 Example in C using reverse communication mode

The KNITRO distribution comes with several C language programs in the directory examples/C. The instructions in examples/C/README.txt explain how to compile and run the examples. This section overviews the coding of driver programs user the reverse communication interface, but the working examples provide more complete detail.

Consider the following nonlinear optimization problem from the Hock and Schittkowski test set [9]:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 100 - (x_2 - x_1^2)^2 + (1 - x_1)^2 && \text{(4.7a)} \\ \text{subject to} \quad & 1 \leq x_1 x_2 && \text{(4.7b)} \\ & 0 \leq x_1 + x_2^2 && \text{(4.7c)} \\ & x_1 \leq 0.5. && \text{(4.7d)} \end{aligned}$$

This problem is coded as examples/C/problemHS15.c.

Every driver starts by allocating a new KNITRO solver instance and checking that it succeeded (`KTR_new()` might return `NULL` if the Ziena license check fails):

```
#include "knitro.h"

/*... Include other headers, define main() ...*/

KTR_context  *kc;

/*... Declare other local variables ...*/

/*---- CREATE A NEW KNITRO SOLVER INSTANCE. */
kc = KTR_new();
if (kc == NULL)
    {
    printf ("Failed to find a Ziena license.\n");
    return( -1 );
    }
```

The next task is to load the problem definition into the solver using `KTR_init_problem()`. The problem has 2 unknowns and 2 constraints, and it is easily seen that all first and second partial derivatives are generally nonzero. The code below captures the problem definition and passes it to KNITRO:

```
/*---- DEFINE PROBLEM SIZES. */
n = 2;
m = 2;
nnzJ = 4;
nnzH = 3;

/*... allocate memory for xLoBnds, xUpBnds, etc. ...*/
```

```
/*---- DEFINE THE OBJECTIVE FUNCTION AND VARIABLE BOUNDS. */
objType = KTR_OBJTYPE_GENERAL;
objGoal = KTR_OBJGOAL_MINIMIZE;
xLoBnds[0] = -KTR_INFBOUND;
xLoBnds[1] = -KTR_INFBOUND;
xUpBnds[0] = 0.5;
xUpBnds[1] = KTR_INFBOUND;

/*---- DEFINE THE CONSTRAINT FUNCTIONS. */
cType[0] = KTR_CONTYPE_QUADRATIC;
cType[1] = KTR_CONTYPE_QUADRATIC;
cLoBnds[0] = 1.0;
cLoBnds[1] = 0.0;
cUpBnds[0] = KTR_INFBOUND;
cUpBnds[1] = KTR_INFBOUND;

/*---- PROVIDE FIRST DERIVATIVE STRUCTURAL INFORMATION. */
jacIndexCons[0] = 0;
jacIndexCons[1] = 0;
jacIndexCons[2] = 1;
jacIndexCons[3] = 1;
jacIndexVars[0] = 0;
jacIndexVars[1] = 1;
jacIndexVars[2] = 0;
jacIndexVars[3] = 1;

/*---- PROVIDE SECOND DERIVATIVE STRUCTURAL INFORMATION. */
hessIndexRows[0] = 0;
hessIndexRows[1] = 0;
hessIndexRows[2] = 1;
hessIndexCols[0] = 0;
hessIndexCols[1] = 1;
hessIndexCols[2] = 1;

/*---- CHOOSE AN INITIAL START POINT. */
xInitial[0] = -2.0;
xInitial[1] =  1.0;

/*---- INITIALIZE KNITRO WITH THE PROBLEM DEFINITION. */
nStatus = KTR_init_problem (kc, n, objGoal, objType,
                            xLoBnds, xUpBnds,
                            m, cType, cLoBnds, cUpBnds,
                            nnzJ, jacIndexVars, jacIndexCons,
                            nnzH, hessIndexRows, hessIndexCols,
                            xInitial, NULL);
if (nStatus != 0)
```

```
    { /*... an error occurred ...*/ }

/*... free xLoBnds, xUpBnds, etc. ...*/
```

Assume for simplicity that the user writes three routines for computing problem information. In examples/C/problemHS15.c these are named `computeFC`, `computeGA`, and `computeH`. To write a driver program using **reverse communications** mode, set up a loop that calls `KTR_solve()` and then computes the requested problem information. The loop continues until `KTR_solve()` returns zero (success), or a negative error code:

```
/*---- SOLVE THE PROBLEM.  IN REVERSE COMMUNICATIONS MODE, KNITRO
 *---- RETURNS WHENEVER IT NEEDS MORE PROBLEM INFO.  THE CALLING
 *---- PROGRAM MUST INTERPRET KNITRO'S RETURN STATUS AND CONTINUE
 *---- SUPPLYING PROBLEM INFORMATION UNTIL KNITRO IS COMPLETE.
 */
while (1)
    {
    nStatus = KTR_solve (kc, x, lambda, evalStatus, &obj, c,
                         objGrad, jac, hess, hvector, NULL);

    if      (nStatus == KTR_RC_EVALFC)
        /*---- KNITRO WANTS obj AND c EVALUATED AT THE POINT x. */
        obj = computeFC (x, c);
    else if (nStatus == KTR_RC_EVALGA)
        /*---- KNITRO WANTS objGrad AND jac EVALUATED AT x. */
        computeGA (x, objGrad, jac);
    else if (nStatus == KTR_RC_EVALH)
        /*---- KNITRO WANTS hess EVALUATED AT (x, lambda). */
        computeH (x, lambda, hess);
    else
        /*---- IN THIS EXAMPLE, OTHER STATUS CODES MEAN KNITRO IS
               FINISHED. */
        break;

    /*---- ASSUME THAT PROBLEM EVALUATION IS ALWAYS SUCCESSFUL.
     *---- IF A FUNCTION OR ITS DERIVATIVE COULD NOT BE EVALUATED
     *---- AT THE GIVEN (x, lambda), THEN SET evalStatus = 1 BEFORE
     *---- CALLING KTR_solve AGAIN. */
    evalStatus = 0;
    }

if (nStatus != KTR_RC_OPTIMAL)
    printf ("KNITRO failed to solve the problem, final status = %d\n",
            nStatus);

/*---- DELETE THE KNITRO SOLVER INSTANCE. */
KTR_free (&kc);
```

This completes the brief overview of creating driver programs to run KNITRO in C using reverse communication mode. Again, more details and options are demonstrated in the programs located in examples/C, including an example for a mixed integer nonlinear programming model. Outputs produced when running KNITRO are discussed in Section 7.

4.4 KNITRO in a C++ application

Calling KNITRO from a C++ application follows the same outline as a C application. The distribution provides a C++ general test harness in the directory examples/C++. In the example, optimization problems are coded as subclasses of an abstract interface and compiled as separate shared objects. A main driver program dynamically loads a problem and sets up **callback mode** (9.8) so KNITRO can invoke the particular problem's evaluation methods. The main driver can also use KNITRO to conveniently check partial derivatives against finite-difference approximations. It is easy to add more test problems to the test environment.

4.5 KNITRO in a Java application

Calling KNITRO from a Java application follows the same outline as a C application. The optimization problem must be defined in terms of arrays and constants that follow the KNITRO API, and then the Java version of `KTR_init_problem()`/`KTR_mip_init_problem()` is called. Java `int` and `double` types map directly to their C counterparts. Having defined the optimization problem, the Java version of `KTR_solve()`/`KTR_mip_solve()` is called in **reverse communications mode** (9.7).

The KNITRO distribution provides a Java Native Interface (JNI) wrapper for the KNITRO callable library functions defined in knitro.h. The Java API loads lib\JNI-knitro.dll, a JNI-enabled form of the KNITRO binary (on Unix the file is called lib/libJNI-knitro.so; on MacIntosh it is lib/libJNI-knitro.jnilib). In this way Java applications can create a KNITRO solver instance and call Java methods that execute KNITRO functions. The JNI form of KNITRO is thread-safe, which means that a Java application can create multiple instances of a KNITRO solver in different threads, each instance solving a different problem. This feature might be important in an application that is deployed on a web server.

To write a Java application, take a look at the sample programs in examples/Java. The call sequence for using KNITRO is almost exactly the same as C applications that call knitro.h functions with a `KTR_context` reference. In Java, an instance of the class `KnitroJava` takes the place of the context reference. The sample programs compile by linking with the Java API class file delivered in the examples/Java/knitrojava.jar archive. This archive also contains the source code for `KnitroJava`. Examine it directly to see the full set of methods supplied with the Java API (not all methods are used in the sample programs). To extract the source code, type the command "`jar xf knitrojava.jar`", and look for com/ziena/knitro/KnitroJava.java.

The sample programs closely mirror the structural form of the C reverse communications example described in Section 4.3. Refer to that section for more information. See Section 4.8 for details on specifying the arrays of partial derivatives that KNITRO needs.

The KNITRO Java API is compiled with Java release 1.5 (see Section 4.7). However, the code does not make use of advanced 1.5 features (for example, there are no generics) and runs equally well on Java release 1.4.

4.6 KNITRO in a Fortran application

Calling KNITRO from a Fortran application follows the same outline as a C application. The optimization problem must be defined in terms of arrays and constants that follow the KNITRO API, and then the Fortran version of `KTR_init_problem()` is called. Fortran `integer` and `double precision` types map directly to C `int` and `double` types. Having defined the optimization problem, the Fortran version of `KTR_solve()` is called in **reverse communications mode** (9.7).

Fortran applications require wrapper functions written in C to (1) isolate the `KTR_context` structure, which has no analog in unstructured Fortran, (2) convert C function names into names recognized by the Fortran linker, and (3) renumber array indices to start from zero (the C convention used by KNITRO) for applications that follow the Fortran convention of starting from one. The wrapper functions can be called from Fortran with exactly the same arguments as their C language counterparts, except for the omission of the `KTR_context` argument.

An example Fortran program and set of C wrappers is provided in examples/Fortran. The code will not be reproduced here, as it closely mirrors the structural form of the C reverse communications example described in Section 4.3. The example loads the matrix sparsity of the optimization problem with indices that start numbering from zero, and therefore requires no conversion from the Fortran convention of numbering from one. The C wrappers provided are sufficient for the simple example, but do not implement all the functionality of the KNITRO callable library. Users are free to write their own C wrapper routines, or extend the example wrappers as needed.

4.7 Compiler specifications

Listed below are the C/C++ compilers used to build KNITRO, and the Java and Fortran compilers used to test programmatic interfaces. It is usually not difficult for Ziena to compile KNITRO in a different environment (for example, it is routinely recompiled to specific versions of `gcc` on Linux). Contact Ziena if your application requires special compilation of KNITRO.

Windows (32-bit x86)
- C/C++: Microsoft Visual Studio C++ 7.1 (.NET 2003 Framework 1.1)
- Java: 1.5.0_16 from Sun
- Fortran: Intel Visual Fortran 9.0

Windows (64-bit x86_64)
- C/C++: Microsoft Visual Studio C++ 8.0 (.NET 2005 Framework 2.0)
- Java: 1.5.0_10 from Sun
- Fortran: Intel Visual Fortran 9.1

Linux (32-bit x86, 64-bit x86_64)
- C/C++: gcc/g++ (compiler version to match the Linux distribution)
- Java: 1.5.0_06 from Sun
- Fortran: g77/g95

Mac OS X (32-bit x86, 64-bit x86_64)
- C/C++: gcc/g++ 4.0.1 (XCode 3.1)
- Java: 1.5.0_24

Solaris 10 (64-bit, x86_64, single-machine licenses only)
- C/C++: gcc/g++ 3.4.3
- Java: 1.5.0_17 from Sun

4.8 Specifying the Jacobian and Hessian matrices

An important issue in using the KNITRO callable library is the ability of the application to specify the Jacobian matrix of the constraints and the Hessian matrix of the Lagrangian function (when using exact Hessians) in sparse form. Below we give an example of how to do this.

Example

Assume we want to use KNITRO to solve the following problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x_0 + x_1 x_2^3 && \text{(4.8a)} \\ \text{subject to} \quad & \cos(x_0) = 0.5 && \text{(4.8b)} \\ & 3 \leq x_0^2 + x_1^2 \leq 8 && \text{(4.8c)} \\ & x_0 + x_1 + x_2 \leq 10 && \text{(4.8d)} \\ & x_0, x_1, x_2 \geq 1. && \text{(4.8e)} \end{aligned}$$

Rewriting in the notation of (4.4), we have

$$\begin{aligned} f(x) &= x_0 + x_1 x_2^3 && \text{(4.9)} \\ c_0(x) &= \cos(x_0) && \text{(4.10)} \\ c_1(x) &= x_0^2 + x_1^2 && \text{(4.11)} \\ c_2(x) &= x_0 + x_1 + x_2. && \text{(4.12)} \\ & && \text{(4.13)} \end{aligned}$$

Computing the Sparse Jacobian Matrix

The gradients (first derivatives) of the objective and constraint functions are given by

$$\nabla f(x) = \begin{bmatrix} 1 \\ x_2^3 \\ 3x_1x_2^2 \end{bmatrix}, \nabla c_0(x) = \begin{bmatrix} -\sin(x_0) \\ 0 \\ 0 \end{bmatrix}, \nabla c_1(x) = \begin{bmatrix} 2x_0 \\ 2x_1 \\ 0 \end{bmatrix}, \nabla c_2(x) = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

The constraint Jacobian matrix $J(x)$ is the matrix whose rows store the (transposed) constraint gradients, i.e.,

$$J(x) = \begin{bmatrix} \nabla c_0(x)^T \\ \nabla c_1(x)^T \\ \nabla c_2(x)^T \end{bmatrix} = \begin{bmatrix} -\sin(x_0) & 0 & 0 \\ 2x_0 & 2x_1 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

In KNITRO, the array `objGrad` stores *all* of the elements of $\nabla f(x)$, while the arrays `jac`, `jacIndexCons`, and `jacIndexVars` store information concerning *only the nonzero* elements of $J(x)$. The array `jac` stores the nonzero values in $J(x)$ evaluated at the current solution estimate x, `jacIndexCons` stores the constraint function (or row) indices corresponding to these values, and `jacIndexVars` stores the variable (or column) indices. There is no restriction on the order in which these elements are stored; however, it is common to store the nonzero elements of $J(x)$ in column-wise fashion. For the example above, the number of nonzero elements `nnzJ` in $J(x)$ is 6, and these arrays are specified as follows in column-wise order.

```
jac[0] = -sin(x[0]);  jacIndexCons[0] = 0; jacIndexVars[0] = 0;
jac[1] = 2*x[0];      jacIndexCons[1] = 1; jacIndexVars[1] = 0;
jac[2] = 1;           jacIndexCons[2] = 2; jacIndexVars[2] = 0;
jac[3] = 2*x[1];      jacIndexCons[3] = 1; jacIndexVars[3] = 1;
jac[4] = 1;           jacIndexCons[4] = 2; jacIndexVars[4] = 1;
jac[5] = 1;           jacIndexCons[5] = 2; jacIndexVars[5] = 2;
```

The values of `jac` depend on the value of x and change during the solution process. The values of `jacIndexCons` and `jacIndexVars` are set in `KTR_init_problem()` and remain constant.

Computing the Sparse Hessian Matrix

The Hessian of the Lagrangian matrix is defined as

$$H(x,\lambda) = \nabla^2 f(x) + \sum_{i=0}^{m-1} \lambda_i \nabla^2 c_i(x), \tag{4.14}$$

where λ is the vector of Lagrange multipliers (dual variables). For the example defined by problem (4.8), The Hessians (second derivatives) of the objective and constraint functions are given by

$$\nabla^2 f(x) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 3x_2^2 \\ 0 & 3x_2^2 & 6x_1x_2 \end{bmatrix}, \quad \nabla^2 c_0(x) = \begin{bmatrix} -\cos(x_0) & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

$$\nabla^2 c_1(x) = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \nabla^2 c_2(x) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Scaling the constraint matrices by their corresponding Lagrange multipliers and summing, we get

$$H(x,\lambda) = \begin{bmatrix} -\lambda_0 \cos(x_0) + 2\lambda_1 & 0 & 0 \\ 0 & 2\lambda_1 & 3x_2^2 \\ 0 & 3x_2^2 & 6x_1x_2 \end{bmatrix}.$$

Since the Hessian matrix will always be a symmetric matrix, KNITRO only stores the nonzero elements corresponding to the upper triangular part (including the diagonal). In the example here, the number of nonzero elements in the upper triangular part of the Hessian matrix `nnzH` is 4. The KNITRO array `hess` stores the values of these elements, while the arrays `hessIndexRows` and `hessIndexCols` store the row and column indices respectively. The order in which these nonzero elements is stored is not important. If we store them column-wise, the arrays `hess`, `hessIndexRows` and `hessIndexCols` are as follows:

```
hess[0] = -lambda[0]*cos(x[0]) + 2*lambda[1];
hessIndexRows[0] = 0;
hessIndexCols[0] = 0;

hess[1] = 2*lambda[1];
hessIndexRows[1] = 1;
hessIndexCols[1] = 1;
```

```
hess[2] = 3*x[2]*x[2];
hessIndexRows[2] = 1;
hessIndexCols[2] = 2;

hess[3] = 6*x[1]*x[2];
hessIndexRows[3] = 2;
hessIndexCols[3] = 2;
```

The values of `hess` depend on the value of x and change during the solution process. The values of `hessIndexRows` and `hessIndexCols` are set in `KTR_init_problem()` and remain constant.

4.9 Determining convexity

Knowing whether or not a function is convex may be useful in methods for mixed integer programming as linearizations derived from convex functions can be used as outer approximations of those constraints. These outer approximations are useful in computing lower bounds. The callable library for the mixed integer programming API allows for the user to specify whether or not the problem functions (objective and constraints) are convex or not. If unknown, they can be marked as such.

A function f is convex if for any two points x and y, we have that

$$f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha)f(y), \text{ for all } \alpha \in [0,1]. \tag{4.15}$$

In identifying the objective or constraints as convex, we are assuming a problem form where the objective is being minimized and the constraints are all formulated as "less than or equal to" constraints. If we are maximizing or looking at "greater than or equal to" constraints, then the objective or constraint should be labeled as convex, if its negation is convex.

More specifically, the objective function $f(x)$ should be marked as convex if when minimizing $f(x)$ satisfies condition (4.15), or if when maximizing $-f(x)$ satisfies condition (4.15). If we consider problem (1.1), a constraint $c_i(x)$ should be labeled as convex if:

- c^L is infinite, c^U is finite and $c_i(x)$ satisfies condition (4.15); or
- c^L is finite, c^U is infinite and $-c_i(x)$ satisfies condition (4.15); or
- c^L is finite, c^U is finite and $c_i(x)$ is linear.

All linear functions are convex. Any nonlinear equality constraint is nonconvex.

The MIP solvers in KNITRO are designed for convex problems (problems where the objective and all the constraints are convex). If one or more functions are nonconvex, these solvers are only heuristics and may terminate at non-optimal points. The continuous solvers in KNITRO can handle either convex or nonconvex models. However, for nonconvex models, they may converge to local (rather than global) optimal solutions.

4.10 Calling without first derivatives

Applications should provide partial first derivatives whenever possible, to make KNITRO more efficient and more robust. If first derivatives cannot be supplied, then the application should instruct KNITRO to calculate finite-difference approximations, as described in Section 9.1. Even though the

application does not evaluate derivatives, it must still provide a sparsity pattern for the constraint Jacobian matrix that specifies which partial derivatives are nonzero. KNITRO uses the sparsity pattern to speed up linear algebra computations. If the sparsity pattern is unknown, then the application should specify a fully dense pattern (i.e., assume all partial derivatives are nonzero).

The code fragment below demonstrates how to define a problem with no derivatives and unknown sparsity pattern. The code is in the C language.

```
/*... define variables, call KTR_new(), etc. ...*/

/*---- DEFINE PROBLEM SIZES.  NOTHING IS KNOWN ABOUT THE DERIVATIVES,
 *---- SO ASSUME THE JACOBIAN IS DENSE.  THIS EXAMPLE HAS 20 VARIABLES
 *---- AND 10 CONSTRAINTS.  NO HESSIAN IS SUPPLIED, SO SET nnzH TO ZERO. */
n = 20;
m = 10;
nnzJ = n * m;
nnzH = 0;


/*... define objType, xLoBnds, xUpBnds, cType, cLoBnds, cUpBnds, etc. ...*/
/*... note that cType is especially useful if constraints are linear */

/*---- DEFINE FIRST DERIVATIVE SPARSITY PATTERN.  NOTHING IS KNOWN
 *---- ABOUT THE DERIVATIVES, SO DEFINE THE JACOBIAN MATRIX TO BE DENSE. */
k = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
    {
        jacIndexCons[k] = j;
        jacIndexVars[k] = i;
        k++;
    }

/*---- INSTRUCT KNITRO TO COMPUTE FIRST DERIVATIVE ESTIMATES
 *---- AND APPROXIMATE THE HESSIAN. */
KTR_set_int_param (kc, KTR_PARAM_GRADOPT, KTR_GRADOPT_CENTRAL);
KTR_set_int_param (kc, KTR_PARAM_HESSOPT, KTR_HESSOPT_LBFGS);

/*---- INITIALIZE KNITRO WITH THE PROBLEM DEFINITION. */
nStatus = KTR_init_problem (kc, n, objGoal, objType,
                            xLoBnds, xUpBnds,
                            m, cType, cLoBnds, cUpBnds,
                            nnzJ, jacIndexVars, jacIndexCons,
                            0, NULL, NULL,
                            NULL, NULL);
if (nStatus != 0)
    { /*... an error occurred ...*/ }
```

```
/*---- SOLVE THE PROBLEM USING REVERSE COMMUNICATIONS MODE.
 *---- KNITRO RETURNS WHENEVER IT NEEDS MORE PROBLEM INFO, WHICH
 *---- IN THIS CASE WILL ONLY BE TO EVALUATE THE OBJECTIVE AND
 *---- CONSTRAINT FUNCTIONS (NO REQUESTS FOR DERIVATIVES).
 *---- MUST PASS objGrad AND jac BECAUSE KNITRO USES THEM TO
 *---- STORE FINITE DIFFERENCE ESTIMATES. */
while (1)
{
    nStatus = KTR_solve (kc, x, lambda, evalStatus, &obj, c,
                         objGrad, jac, NULL, NULL, NULL);

    if      (nStatus == KTR_RC_EVALFC)
        /*---- KNITRO WANTS obj AND c EVALUATED AT THE POINT x. */
        /*... compute obj and c at x...*/
    else
        /*---- IN THIS EXAMPLE, OTHER STATUS CODES MEAN KNITRO IS
               FINISHED. */
        break;
}

if (nStatus != KTR_RC_OPTIMAL)
    printf ("KNITRO failed to solve the problem, final status = %d\n",
            nStatus);

/*---- DELETE THE KNITRO SOLVER INSTANCE. */
KTR_free (&kc);
```

5 User options in KNITRO

KNITRO offers a number of user options for modifying behavior of the solver. Each option takes a value that may be an integer, double precision number, or character string. Options are usually identified by a string name (for example, "`algorithm`"), but programmatic interfaces also identify options by an integer value associated with a C language macro defined in the file knitro.h (for example, "`KTR_PARAM_ALG`"). This section lists all user options in alphabetical order, identified by the string name and the macro definition. User options beginning with `"bar_"` apply only to the barrier/interior-point algorithms; options beginning with `"mip_"` apply only to the mixed integer programming (MIP) solvers; and options specific to the multi-start procedure begin with `"ms_"`.

Sections 5.2 and 5.3 provide instructions on how to set and modify user options.

5.1 Description of KNITRO user options

`algorithm (KTR_PARAM_ALG):` Indicates which algorithm to use to solve the problem (see Section 8).

- `0 (auto):` Let KNITRO automatically choose an algorithm, based on the problem characteristics.
- `1 (direct):` Use the Interior/Direct algorithm.
- `2 (cg):` Use the Interior/CG algorithm.
- `3 (active):` Use the Active Set algorithm.

Default value: `0`

`bar_directinterval (KTR_PARAM_BAR_DIRECTINTERVAL):` Controls the maximum number of consecutive conjugate gradient (CG) steps before KNITRO will try to enforce that a step is taken using direct linear algebra. This option is only valid for the Interior/Direct algorithm and may be useful on problems where KNITRO appears to be taking lots of conjugate gradient steps. Setting `bar_directinterval=0` will try to enforce that only direct steps are taken which may produce better results on some problems.

Default value: `10`

`bar_feasible (KTR_PARAM_BAR_FEASIBLE):` Specifies whether special emphasis is placed on getting and staying feasible in the interior-point algorithms.

- `0 (no):` No special emphasis on feasibility.
- `1 (stay):` Iterates must satisfy inequality constraints once they become sufficiently feasible.
- `2 (get):` Special emphasis is placed on getting feasible before trying to optimize.
- `3 (get_stay):` Implement both options 1 and 2 above.

Default value: `0`

NOTE: This option can only be used with the Interior/Direct and Interior/CG algorithms.

If `bar_feasible=stay` or `bar_feasible=get_stay`, this will activate the feasible version of KNITRO. The feasible version of KNITRO will force iterates to strictly satisfy inequalities, but does not require satisfaction of equality constraints at intermediate iterates (see Section 9.3).

This option and the `honorbnds` option may be useful in applications where functions are undefined outside the region defined by inequalities. The initial point must satisfy inequalities to a *sufficient* degree; if not, KNITRO may generate infeasible iterates and does not switch to the feasible version until a sufficiently feasible point is found. *Sufficient* satisfaction occurs at a point x if it is true for all inequalities that

$$cl + tol \leq c(x) \leq cu - tol \tag{5.16}$$

The constant *tol* is determined by the option `bar_feasmodetol`.

If `bar_feasible=get` or `bar_feasible=get_stay`, KNITRO will place special emphasis on first trying to get feasible before trying to optimize.

See Section 9.3 for more details on this option.

`bar_feasmodetol (KTR_PARAM_BAR_FEASMODETOL)`: Specifies the tolerance in equation (5.16) that determines whether KNITRO will force subsequent iterates to remain feasible. The tolerance applies to all inequality constraints in the problem. This option only has an effect if option `bar_feasible=stay` or `bar_feasible=get_stay`.

Default value: `1.0e-4`

`bar_initmu (KTR_PARAM_BAR_INITMU)`: Specifies the initial value for the barrier parameter μ used with the barrier algorithms. This option has no effect on the Active Set algorithm.

Default value: `1.0e-1`

`bar_initpt (KTR_PARAM_BAR_INITPT)`: Indicates whether an initial point strategy is used with barrier algorithms. This option has no effect on the Active Set algorithm.

`0 (auto)`:	Let KNITRO automatically choose the strategy.
`1 (yes)`:	Shift the initial point to improve barrier algorithm performance.
`2 (no)`:	Do no alter the initial point supplied by the user.

Default value: `0`

`bar_maxbacktrack (KTR_PARAM_BAR_MAXBACKTRACK)`: Indicates the maximum allowable number of backtracks during the linesearch of the Interior/Direct algorithm before reverting to a CG step. Increasing this value will make the Interior/Direct algorithm less likely to take CG steps. If the Interior/Direct algorithm is taking a large number of CG steps (as indicated by a positive value for "*CGits*" in the output), this may improve performance. This option has no effect on the Active Set algorithm.

Default value: `3`

`bar_maxrefactor (KTR_PARAM_BAR_MAXREFACTOR)`: Indicates the maximum number of refactorizations of the KKT system per iteration of the Interior/Direct algorithm before reverting to a CG step. These refactorizations are performed if negative curvature is detected in the model. Rather than reverting to a CG step, the Hessian matrix is modified in an attempt to make the subproblem convex and then the KKT system is refactorized. Increasing this value will make the Interior/Direct algorithm less likely to take CG steps. If the Interior/Direct algorithm is taking a large number of CG steps (as indicated by a positive value for "*CGits*" in the output), this may improve performance. This option has no effect on the Active Set algorithm.

Default value: `0`

`bar_murule (KTR_PARAM_BAR_MURULE)`: Indicates which strategy to use for modifying the barrier parameter μ in the barrier algorithms (see Section 8). Not all strategies are available for both barrier algorithms, as described below. This option has no effect on the Active Set algorithm.

- `0 (auto)`: Let KNITRO automatically choose the strategy.
- `1 (monotone)`: Monotonically decrease the barrier parameter. Available for both barrier algorithms.
- `2 (adaptive)`: Use an adaptive rule based on the complementarity gap to determine the value of the barrier parameter. Available for both barrier algorithms.
- `3 (probing)`: Use a probing (affine-scaling) step to dynamically determine the barrier parameter. Available only for the Interior/Direct algorithm.
- `4 (dampmpc)`: Use a Mehrotra predictor-corrector type rule to determine the barrier parameter, with safeguards on the corrector step. Available only for the Interior/Direct algorithm.
- `5 (fullmpc)`: Use a Mehrotra predictor-corrector type rule to determine the barrier parameter, without safeguards on the corrector step. Available only for the Interior/Direct algorithm.
- `6 (quality)`: Minimize a *quality* function at each iteration to determine the barrier parameter. Available only for the Interior/Direct algorithm.

Default value: `0`

`bar_penaltycons (KTR_PARAM_BAR_PENCONS)`: Indicates whether a penalty approach is applied to the constraints. Using a penalty approach may be helpful when the problem has degenerate or difficult constraints. It may also help to more quickly identify infeasible problems, or achieve feasibility in problems with difficult constraints. This option has no effect on the Active Set algorithm.

- `0 (auto)`: Let KNITRO automatically choose the strategy.
- `1 (none)`: No constraints are penalized.
- `2 (all)`: A penalty approach is applied to all general constraints.

Default value: `0`

`bar_penaltyrule (KTR_PARAM_BAR_PENRULE)`: Indicates which penalty parameter strategy to use for determining whether or not to accept a trial iterate. This option has no effect on the Active Set algorithm.

- `0 (auto)`: Let KNITRO automatically choose the strategy.
- `1 (single)`: Use a single penalty parameter in the merit function to weight feasibility versus optimality.
- `2 (flex)`: Use a more tolerant and flexible step acceptance procedure based on a range of penalty parameter values.

Default value: `0`

`blasoption (KTR_PARAM_BLASOPTION)`: Specifies the BLAS/LAPACK function library to use for basic vector and matrix computations.

- `0 (knitro)`: Use KNITRO built-in functions.
- `1 (intel)`: Use Intel Math Kernel Library (MKL) functions on available platforms.
- `2 (dynamic)`: Use the dynamic library specified with option `blasoptionlib`.

Default value: `1`

NOTE: BLAS and LAPACK functions from Intel Math Kernel Library (MKL) 10.2 are provided with the KNITRO distribution. The MKL is available for Windows (32-bit and 64-bit), Linux (32-bit and 64-bit), and Mac OS X (32-bit and 64-bit); it is *not* available for Solaris. The MKL is not included with the free student edition of KNITRO. On platforms, where the intel MKL is not available, the KNITRO built-in functions are used by default.

BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra PACKage) functions are used throughout KNITRO for fundamental vector and matrix calculations. The CPU time spent in these operations can be measured by setting option `debug=1` and examining the output file kdbg_summ*.txt. Some optimization problems are observed to spend very little CPU time in BLAS/LAPACK operations, while others spend more than 50%. Be aware that the different function implementations can return slightly different answers due to roundoff errors in double precision arithmetic. Thus, changing the value of `blasoption` sometimes alters the iterates generated by KNITRO, or even the final solution point.

The `knitro` option uses built-in BLAS/LAPACK functions based on standard netlib routines (`www.netlib.org`). The `intel` option uses MKL functions written especially for x86 and x86_64 processor architectures. On a machine running an Intel processor (e.g., Pentium 4), testing indicates that the MKL functions can significantly reduce the CPU time in BLAS/LAPACK operations. The `dynamic` option allows users to load any library that implements the functions declared in the file include/blas_lapack.h. Specify the library name with option `blasoptionlib`.

The Intel MKL is provided in the KNITRO lib directory and is loaded at runtime by KNITRO. The operating system's load path must be configured to find this directory or the MKL will fail to load. See Section 5.4 for details.

If your machine uses security enhanced Linux (SELinux), you may see errors when loading the Intel MKL. Refer to Section 2.3 for more information.

`blasoptionlib (KTR_PARAM_BLASOPTIONLIB)`: Specifies a dynamic library name that contains object code for BLAS/LAPACK functions. The library must implement all the functions declared in the file include/blas_lapack.h. The source file blasAcmlExample.c in examples/C provides a wrapper for the AMD Core Math Library (ACML), suitable for machines with an AMD processor. Instructions are given in the file for creating a BLAS/LAPACK dynamic library from the ACML. The operating system's load path must be configured to find the dynamic library, as described in Section 5.4.

NOTE: This option has no effect unless `blasoption=2`.

`cplexlibname (KTR_PARAM_CPLEXLIB)`: See option `lpsolver`.

`debug (KTR_PARAM_DEBUG)`: Controls the level of debugging output. Debugging output can slow execution of KNITRO and should not be used in a production setting. All debugging output is suppressed if option `outlev=0`.

- `0 (none)`: No debugging output.
- `1 (problem)`: Print algorithm information to kdbg*.log output files.
- `2 (execution)`: Print program execution information.

Default value: `0`

`delta (KTR_PARAM_DELTA)`: Specifies the initial trust region radius scaling factor used to determine the initial trust region size.

Default value: `1.0e0`

`feastol (KTR_PARAM_FEASTOL)`: Specifies the final relative stopping tolerance for the feasibility error. Smaller values of `feastol` result in a higher degree of accuracy in the solution with respect to feasibility. See Section 6 for more information.

Default value: `1.0e-6`

`feastol_abs (KTR_PARAM_FEASTOLABS)`: Specifies the final absolute stopping tolerance for the feasibility error. Smaller values of `feastol_abs` result in a higher degree of accuracy in the solution with respect to feasibility. See Section 6 for more information.

Default value: `0.0e0`

`gradopt (KTR_PARAM_GRADOPT)`: Specifies how to compute the gradients of the objective and constraint functions. See Section 9.1 for more information.

- `1 (exact)`: User provides a routine for computing the exact gradients.
- `2 (forward)`: KNITRO computes gradients by forward finite-differences.
- `3 (central)`: KNITRO computes gradients by central finite differences.

Default value: `1`

NOTE: It is highly recommended to provide exact gradients if at all possible as this greatly impacts the performance of the code.

`hessopt (KTR_PARAM_HESSOPT)`: Specifies how to compute the (approximate) Hessian of the Lagrangian. See Section 9.2 for more information.

- `1 (exact)`: User provides a routine for computing the exact Hessian.
- `2 (bfgs)`: KNITRO computes a (dense) quasi-Newton BFGS Hessian.
- `3 (sr1)`: KNITRO computes a (dense) quasi-Newton SR1 Hessian.
- `4 (finite_diff)`: KNITRO computes Hessian-vector products using finite-differences.
- `5 (product)`: User provides a routine to compute the Hessian-vector products.
- `6 (lbfgs)`: KNITRO computes a limited-memory quasi-Newton BFGS Hessian (its size is determined by the option `lmsize`).

Default value: `1`

NOTE: Options `hessopt=4` and `hessopt=5` are not available with the Interior/Direct algorithm.

KNITRO usually performs best when the user provides exact Hessians (`hessopt=1`) or exact Hessian-vector products (`hessopt=5`). If neither can be provided but *exact* gradients are available (i.e., `gradopt=1`), then `hessopt=4` is recommended. This option is comparable in terms of robustness to the exact Hessian option and typically not much slower in terms of time, provided that gradient evaluations are not a dominant cost. If exact gradients cannot be provided, then one of the quasi-Newton options is preferred. Options `hessopt=2` and `hessopt=3` are only recommended for small problems ($n < 1000$) since they require working with a dense Hessian approximation. Option `hessopt=6` should be used for large problems. See Section 9.2 for more information.

`honorbnds (KTR_PARAM_HONORBNDS)`: Indicates whether or not to enforce satisfaction of simple variable bounds throughout the optimization (see Section 9.4). This option and the `bar_feasible` option may be useful in applications where functions are undefined outside the region defined by inequalities.

- `0 (no)`: KNITRO does not require that the bounds on the variables be satisfied at intermediate iterates.
- `1 (always)`: KNITRO enforces that the initial point and all subsequent solution estimates satisfy the bounds on the variables.
- `2 (initpt)`: KNITRO enforces that the initial point satisfies the bounds on the variables.

Default value: `2`

`infeastol (KTR_PARAM_INFEASTOL)`: Specifies the (relative) tolerance used for declaring infeasibility of a model. Smaller values of `infeastol` make it more difficult to satisfy the conditions KNITRO uses for detecting infeasible models. If you believe KNITRO incorrectly declares a model to be infeasible, then you should try a smaller value for `infeastol`.

Default value: `1.0e-8`

`linsolver (KTR_PARAM_LINSOLVER)`: Indicates which linear solver to use to solve linear systems arising in KNITRO algorithms.

- `0 (auto)`: Let KNITRO automatically choose the linear solver.
- `1 (internal)`: Not currently used; reserved for future use. Same as `auto` for now.
- `2 (hybrid)`: Use a hybrid approach where the solver chosen depends on the particular linear system which needs to be solved.
- `3 (qr)`: Use a dense QR method. This approach uses LAPACK QR routines. Since it uses a dense method, it is only efficient for small problems. It may often be the most efficient method for small problems with dense Jacobians or Hessian matrices.
- `4 (ma27)`: Use the HSL MA27 sparse symmetric indefinite solver.
- `5 (ma57)`: Use the HSL MA57 sparse symmetric indefinite solver.

Default value: 0

NOTE: The QR linear solver and the HSL MA57 linear solver both make frequent use of Basic Linear Algebra Subroutines (BLAS) for internal linear algebra operations. If using option `linsolver=qr` or `linsolver=ma57`, it is *highly* recommended to use *optimized* BLAS for your particular machine. This can result in dramatic speedup. On Windows, Linux and Mac OS X platforms, KNITRO provides the Intel Math Kernel Library (MKL) BLAS in the lib folder of the KNITRO distribution. This BLAS library is optimized for Intel processors and can be selected by setting `blasoption=intel`. Please read the notes under the `blasoption` user option in this section for more details about the BLAS options in KNITRO and how to make sure that the Intel MKL BLAS or other user-specified BLAS can be loaded at runtime by KNITRO.

`lmsize (KTR_PARAM_LMSIZE)`: Specifies the number of limited memory pairs stored when approximating the Hessian using the limited-memory quasi-Newton BFGS option. The value must be between 1 and 100 and is only used with `hessopt=6`. Larger values may give a more accurate, but more expensive, Hessian approximation. Smaller values may give a less accurate, but faster, Hessian approximation. When using the limited memory BFGS approach it is recommended to experiment with different values of this parameter. See Section 9.2 for more details.

Default value: 10

`lpsolver (KTR_PARAM_LPSOLVER)`: Indicates which linear programming simplex solver the KNITRO Active Set algorithm uses when solving internal LP subproblems. This option has no effect on the Interior/Direct and Interior/CG algorithms.

`1 (internal)`: KNITRO uses its default LP solver.

`2 (cplex)`: KNITRO uses IBM ILOG-CPLEX, provided the user has a valid CPLEX license. The CPLEX library is loaded dynamically after `KTR_solve()` is called.

Default value: 1

If `lpsolver=cplex` then the CPLEX shared object library or DLL must reside in the operating system's load path (see Section 5.4). If this option is selected, KNITRO will automatically look for (in order): CPLEX 11.2, CPLEX 11.1, CPLEX 11.0, CPLEX 10.2, CPLEX 10.1, CPLEX 10.0, CPLEX 9.1, CPLEX 9.0, or CPLEX 8.0.

To override the automatic search and load a particular CPLEX library, set its name with the character type user option `cplexlibname`. Either supply the full path name in this option, or make sure the library resides in a directory that is listed in the operating system's load path (see Section 5.4). For example, to specifically load the Windows CPLEX library `cplex90.dll`, make sure the directory containing the library is part of the `PATH` environment variable, and call the following (also be sure to check the return status of this call):

```
KTR_set_char_param_by_name (kc, "cplexlibname", "cplex90.dll");
```

`maxcgit (KTR_PARAM_MAXCGIT)`: Determines the maximum allowable number of inner conjugate gradient (CG) iterations per KNITRO minor iteration.

`0`: Let KNITRO automatically choose a value based on the problem size.

n: At most $n > 0$ CG iterations may be performed during one minor iteration of KNITRO.

Default value: `0`

`maxcrossit (KTR_PARAM_MAXCROSSIT)`: Specifies the maximum number of crossover iterations before termination. If the value is positive and the algorithm in operation is Interior/Direct or Interior/CG, then KNITRO will crossover to the Active Set algorithm near the solution. The Active Set algorithm will then perform at most `maxcrossit` iterations to get a more exact solution. If the value is 0, no Active Set crossover occurs and the interior-point solution is the final result.

If Active Set crossover is unable to improve the approximate interior-point solution, then KNITRO will restore the interior-point solution. In some cases (especially on large-scale problems or difficult degenerate problems) the cost of the crossover procedure may be significant – for this reason, crossover is disabled by default. Enabling crossover generally provides a more accurate solution than Interior/Direct or Interior/CG. See Section 9.5 for more information.

Default value: `0`

`maxit (KTR_PARAM_MAXIT)`: Specifies the maximum number of iterations before termination.

`0`: Let KNITRO automatically choose a value based on the problem type. Currently KNITRO sets this value to 10000 for LPs/NLPs and 3000 for MIP problems.

n: At most $n > 0$ iterations may be performed before terminating.

Default value: `0`

`maxtime_cpu (KTR_PARAM_MAXTIMECPU)`: Specifies, in seconds, the maximum allowable CPU time before termination.

Default value: `1.0e8`

`maxtime_real (KTR_PARAM_MAXTIMEREAL)`: Specifies, in seconds, the maximum allowable real time before termination.

Default value: `1.0e8`

`mip_branchrule (KTR_PARAM_MIP_BRANCHRULE)`: Specifies which branching rule to use for MIP branch and bound procedure.

`0 (auto)`: Let KNITRO automatically choose the branching rule.

`1 (most_frac)`: Use most fractional (most infeasible) branching.

`2 (pseudocost)`: Use pseudo-cost branching.

`3 (strong)`: Use strong branching (see options `mip_strong_candlim`, `mip_strong_level` and `mip_strong_maxit` for further control of strong branching procedure).

Default value: `0`

`mip_debug (KTR_PARAM_MIP_DEBUG)`: Specifies debugging level for MIP solution.

`0 (none):` No MIP debugging output created.

`1 (all):` Write MIP debugging output to the file kdbg_mip.log.

Default value: `0`

`mip_gub_branch (KTR_PARAM_MIP_GUB_BRANCH):` Specifies whether or not to branch on generalized upper bounds (GUBs).

`0 (no):` Do not branch on GUBs.

`1 (yes):` Allow branching on GUBs.

Default value: `0`

`mip_heuristic (KTR_PARAM_MIP_HEURISTIC):` Specifies which MIP heuristic search approach to apply to try to find an initial integer feasible point. If a heuristic search procedure is enabled, it will run for at most `mip_heuristic_maxit` iterations, before starting the branch and bound procedure.

`0 (auto):` Let KNITRO choose the heuristic to apply (if any).

`1 (none):` No heuristic search applied.

`2 (feaspump):` Apply feasibility pump heuristic.

`3 (mpec):` Apply heuristic based on MPEC formulation.

Default value: `0`

`mip_heuristic_maxit (KTR_PARAM_MIP_HEURISTIC_MAXIT):` Specifies the maximum number of iterations to allow for MIP heuristic, if one is enabled.

Default value: `100`

`mip_implications (KTR_PARAM_MIP_IMPLICATNS):` Specifies whether or not to add constraints to the MIP derived from logical implications.

`0 (no):` Do not add constraints from logical implications.

`1 (yes):` KNITRO adds constraints from logical implications.

Default value: `1`

`mip_integer_tol (KTR_PARAM_MIP_INTEGERTOL):` This value specifies the threshold for deciding whether or not a variable is determined to be an integer.

Default value: `1.0e-8`

`mip_integral_gap_abs (KTR_PARAM_MIP_INTGAPABS):` The absolute integrality gap stop tolerance for MIP. See Section 6.2 for more information.

Default value: `1.0e-6`

`mip_integral_gap_rel (KTR_PARAM_MIP_INTGAPREL):` The relative integrality gap stop tolerance for MIP. See Section 6.2 for more information.

Default value: `1.0e-6`

`mip_knapsack (KTR_PARAM_MIP_KNAPSACK)`: Specifies rules for adding MIP knapsack cuts.

- `0 (none)`: Do not add knapsack cuts.
- `1 (ineqs)`: Add cuts derived from inequalities only.
- `2 (ineqs_eqs)`: Add cuts derived from both inequalities and equalities.

Default value: `1`

`mip_lpalg (KTR_PARAM_MIP_LPALG)`: Specifies which algorithm to use for any linear programming (LP) subproblem solves that may occur in the MIP branch and bound procedure. LP subproblems may arise if the problem is a mixed integer linear program (MILP), or if using `mip_method=HQG`. (Nonlinear programming subproblems use the algorithm specified by the `algorithm` option.)

- `0 (auto)`: Let KNITRO automatically choose an algorithm, based on the problem characteristics.
- `1 (direct)`: Use the Interior/Direct (barrier) algorithm.
- `2 (cg)`: Use the Interior/CG (barrier) algorithm.
- `3 (active)`: Use the Active Set (simplex) algorithm.

Default value: `0`

`mip_maxnodes (KTR_PARAM_MIP_MAXNODES)`: Specifies the maximum number of nodes explored (0 means no limit).

Default value: `100000`

`mip_maxsolves (KTR_PARAM_MIP_MAXSOLVES)`: Specifies the maximum number of subproblem solves allowed (0 means no limit).

Default value: `200000`

`mip_maxtime_cpu (KTR_PARAM_MIP_MAXTIMECPU)`: Specifies the maximum allowable CPU time in seconds for the complete MIP solution. Use `maxtime_cpu` to additionally limit time spent per subproblem solve.

Default value: `1.0e8`

`mip_maxtime_real (KTR_PARAM_MIP_MAXTIMEREAL)`: Specifies the maximum allowable real time in seconds for the complete MIP solution. Use `maxtime_real` to additionally limit time spent per subproblem solve.

Default value: `1.0e8`

`mip_method (KTR_PARAM_MIP_METHOD)`: Specifies which MIP method to use.

- `0 (auto)`: Let KNITRO automatically choose the method.
- `1 (BB)`: Use the standard branch and bound method.
- `2 (HQG)`: Use the hybrid Quesada-Grossman method (for convex, nonlinear problems only).

Default value: 0

mip_outinterval (KTR_PARAM_MIP_OUTINTERVAL): Specifies node printing interval for mip_outlevel when mip_outlevel>0.

- 1: Print output every node.
- 2: Print output every 2nd node.
- N: Print output every Nth node.

Default value: 10

mip_outlevel (KTR_PARAM_MIP_OUTLEVEL): Specifies how much MIP information to print.

- 0 (none): Do not print any MIP node information.
- 1 (iters): Print one line of output for every node.

Default value: 1

mip_outsub (KTR_PARAM_MIP_OUTSUB): Specifies MIP subproblem solve debug output control. This output is only produced if mip_debug=1 and appears in the file kdbg_mip.log

- 0: Do not print any debug output from subproblem solves.
- 1: Subproblem debug output enabled, controlled by option outlev.
- 2: Subproblem debug output enabled and print problem characteristics.

Default value: 0

mip_pseudoinit (KTR_PARAM_MIP_PSEUDOINIT): Specifies the method used to initialize pseudo-costs corresponding to variables that have not yet been branched on in the MIP method.

- 0: Let KNITRO automatically choose the method.
- 1: Initialize using the average value of computed pseudo-costs.
- 2: Initialize using strong branching.

Default value: 0

mip_rootalg (KTR_PARAM_MIP_ROOTALG): Specifies which algorithm to use for the root node solve in MIP (same options as algorithm user option). *Default value*: 0

mip_rounding (KTR_PARAM_MIP_ROUNDING): Specifies the MIP rounding rule to apply.

- 0 (auto): Let KNITRO choose the rounding rule.
- 1 (none): Do not round if a node is infeasible.
- 2 (heur_only): Round using a fast heuristic only.
- 3 (nlp_sometimes): Round and solve a subproblem if likely to succeed.
- 4 (nlp_always): Always round and solve a subproblem.

Default value: 0

`mip_selectrule (KTR_PARAM_MIP_SELECTRULE)`: Specifies the MIP select rule for choosing the next node in the branch and bound tree.

- `0 (auto)`: Let KNITRO choose the node selection rule.
- `1 (depth_first)`: Search the tree using a depth first procedure.
- `2 (best_bound)`: Select the node with the best relaxation bound.
- `3 (combo_1)`: Use depth first unless pruned, then best bound.

Default value: `0`

`mip_strong_candlim (KTR_PARAM_MIP_STRONG_CANDLIM)`: Specifies the maximum number of candidates to explore for MIP strong branching. *Default value*: `10`

`mip_strong_level (KTR_PARAM_MIP_STRONG_LEVEL)`: Specifies the maximum number of tree levels on which to perform MIP strong branching. *Default value*: `10`

`mip_strong_maxit (KTR_PARAM_MIP_STRONG_MAXIT)`: Specifies the maximum number of iterations to allow for MIP strong branching solves. *Default value*: `1000`

`mip_terminate (KTR_PARAM_MIP_TERMINATE)`: Specifies conditions for terminating the MIP algorithm.

- `0 (optimal)`: Terminate at optimum (see Section 6 for more information).
- `1 (feasible)`: Terminate at first integer feasible point.

Default value: `0`

`ms_enable or multistart (KTR_PARAM_MULTISTART)`: Indicates whether KNITRO will solve from multiple start points to find a better local minimum. See Section 9.6 for details.

- `0 (no)`: KNITRO solves from a single initial point.
- `1 (yes)`: KNITRO solves using multiple start points.

Default value: `0`

`ms_maxbndrange (KTR_PARAM_MSMAXBNDRANGE)`: Specifies the maximum range that an unbounded variable can take when determining new start points. If a variable is unbounded in one or both directions, then new start point values are restricted by the option. If x_i is such a variable, then all initial values satisfy

$$\max\{b_i^L, x_i^0 - \texttt{ms_maxbndrange}/2\} \leq x_i \leq \min\{b_i^U, x_i^0 + \texttt{ms_maxbndrange}/2\},$$

where x_i^0 is the initial value of x_i provided by the user, and b_i^L and b_i^U are the variable bounds (possibly infinite) on x_i. This option has no effect unless `ms_enable=yes`.

Default value: `1000.0`

`ms_maxsolves (KTR_PARAM_MSMAXSOLVES)`: Specifies how many start points to try in multi-start. This option has no effect unless `ms_enable=yes`.

- `0`: Let KNITRO automatically choose a value based on the problem size. The value is $\min(200, 10N)$, where N is the number of variables in the problem.

n: Try $n > 0$ start points.

Default value: `0`

`ms_maxtime_cpu (KTR_PARAM_MSMAXTIMECPU)`: Specifies, in seconds, the maximum allowable CPU time before termination. The limit applies to the operation of KNITRO since multi-start began; in contrast, the value of `maxtime_cpu` limits how long KNITRO iterates from a single start point. Therefore, `ms_maxtime_cpu` should be greater than `maxtime_cpu`. This option has no effect unless `ms_enable=yes`.

Default value: `1.0e8`

`ms_maxtime_real (KTR_PARAM_MSMAXTIMEREAL)`: Specifies, in seconds, the maximum allowable real time before termination. The limit applies to the operation of KNITRO since multi-start began; in contrast, the value of `maxtime_real` limits how long KNITRO iterates from a single start point. Therefore, `ms_maxtime_real` should be greater than `maxtime_real`. This option has no effect unless `ms_enable=yes`.

Default value: `1.0e8`

`ms_num_to_save (KTR_PARAM_MSNUMTOSAVE)`: Specifies the number of distinct feasible points to save in a file named knitro_mspoints.log. Each point results from a KNITRO solve from a different starting point, and must satisfy the absolute and relative feasibility tolerances. The file stores points in order from best objective to worst. Points are distinct if they differ in objective value or some component by the value of `ms_savetol` using a relative tolerance test (see Section 9.6). This option has no effect unless `ms_enable=yes`.

Default value: `0`

`ms_savetol (KTR_PARAM_MSSAVETOL)`: Specifies the tolerance for deciding if two feasible points are distinct. Points are distinct if they differ in objective value or some component by the value of `ms_savetol` using a relative tolerance test (see Section 9.6). A large value can cause the saved feasible points in the file knitro_mspoints.log to cluster around more widely separated points. This option has no effect unless `ms_enable=yes` and `ms_num_to_save` is positive.

Default value: `1.0e-6`

`ms_startptrange (KTR_PARAM_MSSTARTPTRANGE)`: Specifies the maximum range that each variable can take when determining new start points. If a variable has upper and lower bounds and the difference between them is less than `ms_startptrange`, then new start point values for the variable can be any number between its upper and lower bounds. If the variable is unbounded in one or both directions, or the difference between bounds is greater than the minimum of `ms_startptrange` and `ms_maxbndrange`, then new start point values are restricted by the option. If x_i is such a variable, then all initial values satisfy

$$\max\{b_i^L, x_i^0 - \tau\} \leq x_i \leq \min\{b_i^U, x_i^0 + \tau\},$$

$$\tau = \min\{\texttt{ms_startptrange}/2, \texttt{ms_maxbndrange}/2\}$$

where x_i^0 is the initial value of x_i provided by the user, and b_i^L and b_i^U are the variable bounds (possibly infinite) on x_i. This option has no effect unless `ms_enable=yes`.

Default value: `1.0e20`

`ms_terminate (KTR_PARAM_MSTERMINATE)`: Specifies the condition for terminating multi-start. This option has no effect unless `ms_enable=yes`.

`0`:	Terminate after `ms_maxsolves`.
`1`:	Terminate after the first local optimal solution is found or `ms_maxsolves`, whichever comes first.
`2`:	Terminate after the first feasible solution estimate is found or `ms_maxsolves`, whichever comes first.

Default value: `0`

`newpoint (KTR_PARAM_NEWPOINT)`: Specifies additional action to take after every iteration in a solve of a continuous problem. An iteration of KNITRO results in a new point that is closer to a solution. The new point includes values of `x` and Lagrange multipliers `lambda`. The "newpoint" feature in KNITRO is currently only available for continuous problems (solved via `KTR_solve()`).

`0 (none)`:	KNITRO takes no additional action.
`1 (saveone)`:	KNITRO writes `x` and `lambda` to the file knitro_newpoint.log. Previous contents of the file are overwritten.
`2 (saveall)`:	KNITRO appends `x` and `lambda` to the file knitro_newpoint.log. *Warning:* this option can generate a very large file. All iterates, including the start point, crossover points, and the final solution are saved. Each iterate also prints the objective value at the new point, except the initial start point.
`3 (user)`:	If using callback mode (see Section 9.8) and a user callback function is defined with `KTR_set_newpoint_callback()`, then KNITRO will invoke the callback function after every iteration. If using reverse communications mode (see Section 9.7), then KNITRO will return to the driver level after every iteration with `KTR_solve()` returning the integer value defined by `KTR_RC_NEWPOINT` (6).

Default value: `0`

`objrange (KTR_PARAM_OBJRANGE)`: Specifies the extreme limits of the objective function for purposes of determining unboundedness. If the magnitude of the objective function becomes greater than `objrange` for a feasible iterate, then the problem is determined to be unbounded and KNITRO proceeds no further.

Default value: `1.0e20`

`opttol (KTR_PARAM_OPTTOL)`: Specifies the final relative stopping tolerance for the KKT (optimality) error. Smaller values of `opttol` result in a higher degree of accuracy in the solution with respect to optimality. See Section 6 for more information.

Default value: `1.0e-6`

`opttol_abs (KTR_PARAM_OPTTOLABS)`: Specifies the final absolute stopping tolerance for the KKT (optimality) error. Smaller values of `opttol_abs` result in a higher degree of accuracy in the solution with respect to optimality. See Section 6 for more information.

Default value: `0.0e0`

`outappend (KTR_PARAM_OUTAPPEND)`: Specifies whether output should be started in a new file, or appended to existing files. The option affects knitro.log and files produced when `debug=1`. It does not affect knitro_newpoint.log, which is controlled by option `newpoint`.

- `0 (no)`: Erase any existing files when opening for output.
- `1 (yes)`: Append output to any existing files.

Default value: `0`

NOTE: The option should not be changed after calling `KTR_init_problem()`.

`outdir (KTR_PARAM_OUTDIR)`: Specifies a single directory as the location to write all output files. The option should be a full pathname to the directory, and the directory must already exist.

NOTE: The option should not be changed after calling `KTR_init_problem()` or `KTR_mip_init_problem()`.

`outlev (KTR_PARAM_OUTLEV)`: Controls the level of output produced by KNITRO.

- `0 (none)`: Printing of all output is suppressed.
- `1 (summary)`: Print only summary information.
- `2 (iter_10)`: Print basic information every 10 iterations.
- `3 (iter)`: Print basic information at each iteration.
- `4 (iter_verbose)`: Print basic information and the function count at each iteration.
- `5 (iter_x)`: Print all the above, and the values of the solution vector `x`.
- `6 (all)`: Print all the above, and the values of the constraints `c` at `x` and the Lagrange multipliers `lambda`.

Default value: `2`

`outmode (KTR_PARAM_OUTMODE)`: Specifies where to direct the output from KNITRO.

- `0 (screen)`: Output is directed to standard out (e.g., screen).
- `1 (file)`: Output is sent to a file named knitro.log.
- `2 (both)`: Output is directed to both the screen and file knitro.log.

Default value: `0`

`pivot (KTR_PARAM_PIVOT)`: Specifies the initial pivot threshold used in factorization routines. The value should be in the range [0 .. 0.5] with higher values resulting in more pivoting (more stable factorizations). Values less than 0 will be set to 0 and values larger than 0.5 will be set to 0.5. If `pivot` is non-positive, initially no pivoting will be performed. Smaller values may improve the speed of the code but higher values are recommended for more stability (for example, if the problem appears to be very ill-conditioned).

Default value: `1.0e-8`

`scale (KTR_PARAM_SCALE)`: Performs a scaling of the objective and constraint functions based on their values at the initial point. If scaling is performed, all internal computations, including the stopping tests, are based on the scaled values.

`0 (no):` No scaling is performed.

`1 (yes):` KNITRO is allowed to scale the objective function and constraints.

Default value: `1`

`soc (KTR_PARAM_SOC):` Specifies whether or not to try second order corrections (SOC). A second order correction may be beneficial for problems with highly nonlinear constraints.

`0 (no):` No second order correction steps are attempted.

`1 (maybe):` Second order correction steps may be attempted on some iterations.

`2 (yes):` Second order correction steps are always attempted if the original step is rejected and there are nonlinear constraints.

Default value: `1`

`xtol (KTR_PARAM_XTOL):` The optimization process will terminate if the relative change in all components of the solution point estimate is less than `xtol`. If using the Interior/Direct or Interior/CG algorithm and the barrier parameter is still large, KNITRO will first try decreasing the barrier parameter before terminating.

Default value: `1.0e-15`

5.2 The KNITRO options file

The KNITRO options file allows the user to easily change user options by editing a text file, instead of modifying application code. (Note that the AMPL interface to KNITRO cannot read such a file. Other modeling environments may be able to read an options file – please check with the modeling vendor.)

Options are set by specifying a keyword and a corresponding value on a line in the options file. Lines that begin with a "`#`" character are treated as comments and blank lines are ignored. For example, to set the maximum allowable number of iterations to 500, you could create the following options file:

```
# KNITRO Options file
maxit     500
```

The options file is read into KNITRO by calling the following function before invoking `KTR_solve()` or `KTR_mip_solve()`:

```
int KTR_load_param_file (KTR_context *kc, char const *filename)
```

For example, if the options file is named myoptions.opt:

```
status = KTR_load_param_file (kc, "myoptions.opt");
```

The full set of options used by KNITRO in a given solve may be written to a text file through the function call:

```
int KTR_save_param_file (KTR_context *kc, char const *filename)
```

For example:

```
status = KTR_save_param_file (kc, "knitro.opt");
```

A sample options file knitro.opt is provided for convenience and can be found in the examples/C directory. Note that this file is only read by application drivers that call `KTR_load_param_file()`, such as examples/C/callbackExample2.c.

Most user options can be specified with either a numeric value or a string value. The individual user options and their possible numeric values are described in Section 5.1. String values are listed in the comments of the file examples/C/knitro.opt provided with the distribution.

5.3 Setting options through function calls

The functions for setting user options have the form:

```
int KTR_set_int_param (KTR_context *kc, int param_id, int value)
```

for setting integer valued parameters, or

```
int KTR_set_double_param (KTR_context *kc, int param_id, double value)
```

for setting double precision valued parameters.

For example, to specify the Interior/CG algorithm and a tight optimality stop tolerance:

```
status = KTR_set_int_param (kc, KTR_PARAM_ALG, KTR_ALG_BAR_CG);
status = KTR_set_double_param (kc, KTR_PARAM_OPTTOL, 1.0e-8);
```

NOTE: User parameters cannot be set after beginning the optimization process; i.e., after making the first call to `KTR_solve()`/`KTR_mip_solve()`. Some options cannot be set after calling `KTR_init_problem()`/`KTR_mip_init_problem()`.

5.4 Loading dynamic libraries

Some user options instruct KNITRO to load dynamic libraries at runtime. This will not work unless the executable can find the desired library using the operating system's *load path*. Usually this is done by appending the path to the directory that contains the library to an environment variable. For example, suppose the library to be loaded is in the KNITRO lib directory. The instructions below will correctly modify the load path.

On Windows, type (assuming KNITRO 7.0.0 is installed at its default location)

```
> set PATH=%PATH%;C:\Program Files\Ziena\knitro-7.0.0-z\lib
```

On Mac OS X, type (assuming KNITRO 7.0.0 is installed at `/tmp`)

```
> export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:/tmp/knitro-7.0.0-z/lib
```

If you run a Unix bash shell, then type (assuming KNITRO 7.0.0 is installed at `/tmp`)

```
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/tmp/knitro-7.0.0-z/lib
```

If you run a Unix csh or tcsh shell, then type (assuming KNITRO 7.0.0 is installed at `/tmp`)

```
> setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/tmp/knitro-7.0.0-z/lib
```

6 KNITRO termination test and optimality

6.1 Continuous problems

The first-order conditions for identifying a locally optimal solution of the problem (1.1) are:

$$\nabla_x \mathcal{L}(x,\lambda) = \nabla f(x) + \sum_{i=1..m} \lambda_i^c \nabla c_i(x) + \sum_{j=1..n} \lambda_j^b = 0 \tag{6.17}$$

$$\lambda_i^c \min[(c_i(x) - c_i^L), (c_i^U - c_i(x))] = 0, \quad i = 1..m \tag{6.18}$$

$$\lambda_j^b \min[(x_j - b_j^L), (b_j^U - x_j)] = 0, \quad j = 1..n \tag{6.19}$$

$$c_i^L \le c_i(x) \le c_i^U, \quad i = 1..m \tag{6.20}$$

$$b_j^L \le x_j \le b_j^U, \quad j = 1..n \tag{6.21}$$

$$\lambda_i^c \ge 0, \quad i \in \mathcal{I},\ c_i^L \text{infinite},\ c_i^U \text{finite} \tag{6.22}$$

$$\lambda_i^c \le 0, \quad i \in \mathcal{I},\ c_i^U \text{infinite},\ c_i^L \text{finite} \tag{6.23}$$

$$\lambda_j^b \ge 0, \quad j \in \mathcal{B},\ b_j^L \text{infinite},\ b_j^U \text{finite} \tag{6.24}$$

$$\lambda_j^b \le 0, \quad j \in \mathcal{B},\ b_j^U \text{infinite},\ b_j^L \text{finite}. \tag{6.25}$$

Here $\mathcal{I}$ and $\mathcal{B}$ represent the sets of indices corresponding to the general inequality constraints and (non-fixed) variable bound constraints respectively. In the conditions above, λ_i^c is the Lagrange multiplier corresponding to constraint $c_i(x)$, and λ_j^b is the Lagrange multiplier corresponding to the simple bounds on the variable x_j. There is exactly one Lagrange multiplier for each constraint and variable. The Lagrange multiplier may be restricted to take on a particular sign depending on whether the corresponding constraint (or variable) is upper bounded or lower bounded as indicated in (6.22)–(6.25). If the constraint (or variable) has both a finite lower and upper bound, then the appropriate sign of the multiplier depends on which bound (if either) is binding (active) at the solution.

In KNITRO we define the feasibility error (`FeasErr`) at a point x^k to be the maximum violation of the constraints (6.20), (6.21), i.e.,

$$\text{FeasErr} = \max_{i=1..m, j=1..n} (0, (c_i^L - c_i(x^k)), (c_i(x^k) - c_i^U), (b_j^L - x_j^k), (x_j^k - b_j^U)), \tag{6.26}$$

while the optimality error (`OptErr`) is defined as the maximum violation of the first three conditions (6.17)–(6.19). The remaining conditions on the sign of the multipliers (6.22)–(6.25) are enforced explicitly throughout the optimization. In order to take into account problem scaling in the termination test, the following scaling factors are defined

$$\tau_1 = \max(1, (c_i^L - c_i(x^0)), (c_i(x^0) - c_i^U), (b_j^L - x_j^0), (x_j^0 - b_j^U)), \tag{6.27}$$

$$\tau_2 = \max(1, \|\nabla f(x^k)\|_\infty), \tag{6.28}$$

where x^0 represents the initial point.

For unconstrained problems, the scaling (6.28) is not effective since $\|\nabla f(x^k)\|_\infty \to 0$ as a solution is approached. Therefore, for unconstrained problems only, the following scaling is used in the termination test

$$\tau_2 = \max(1, \min(|f(x^k)|, \|\nabla f(x^0)\|_\infty)), \tag{6.29}$$

in place of (6.28).

KNITRO stops and declares `Locally optimal solution found` if the following stopping conditions are satisfied:

$$\text{FeasErr} \leq \max(\tau_1 * \texttt{feastol}, \texttt{feastol_abs}) \tag{6.30}$$
$$\text{OptErr} \leq \max(\tau_2 * \texttt{opttol}, \texttt{opttol_abs}) \tag{6.31}$$

where `feastol`, `opttol`, `feastol_abs`, and `opttol_abs` are constants defined by user options (see Section 5).

This stopping test is designed to give the user much flexibility in deciding when the solution returned by KNITRO is accurate enough. One can use a scaled stopping test (which is the recommended default option) by setting `feastol_abs` and `opttol_abs` equal to `0.0e0`. Likewise, an absolute stopping test can be enforced by setting `feastol` and `opttol` equal to `0.0e0`.

Note that the stopping conditions (6.30),(6.31) apply to the problem being solved internally by KNITRO. If the user option `scale=yes` (see Section 5.1), then the problem objective and constraint functions may first be scaled before the problem is sent to KNITRO for the optimization. In this case, the stopping conditions apply to the scaled form of the problem. If the accuracy achieved by KNITRO with the default settings is not satisfactory, the user may either decrease the tolerances described above, or try setting `scale=no`.

Unbounded problems

Since by default, KNITRO uses a relative/scaled stopping test it is possible for the optimality conditions to be satisfied within the tolerances given by (6.30)-(6.31) for an unbounded problem. For example, if $\tau_2 \to \infty$ while the optimality error stays bounded, condition (6.31) will eventually be satisfied for some `opttol>0`. If you suspect that your problem may be unbounded, using an absolute stopping test will allow KNITRO to detect this.

6.2 Discrete or mixed integer problems

Algorithms for solving versions of (1.1) where one or more of the variables are restricted to take on only discrete values, proceed by solving a sequence of continuous relaxations, where the discrete variables are *relaxed* such that they can take on any continuous value. The *global* solutions, $f(x_R)$, of these relaxed problems provide a lower bound on the optimal objective value for problem (1.1) (upper bound if maximizing). If a feasible point is found for problem (1.1) that satisfies the discrete restrictions on the variables, then this provides an upper bound on the optimal objective value of problem (1.1) (lower bound if maximizing). We will refer to these feasible points as *incumbent* points and denote the objective value at an incumbent point by $f(x_I)$. Assuming all the continuous subproblems have been solved to global optimality (if the problem is convex, all local solutions are global solutions), an optimal solution of problem (1.1) is verified when the lower bound and upper bound are equal.

KNITRO declares optimality for a discrete problem when the gap between the best (i.e., largest) lower bound $f^*(x_R)$ and the best (i.e., smallest) upper bound $f^*(x_I)$ is less than a threshold determined by the user options, `mip_integral_gap_abs` and `mip_integral_gap_rel`. Specifically, KNITRO declares optimality when either

$$f^*(x_I) - f^*(x_R) \leq \texttt{mip_integral_gap_abs}, \tag{6.32}$$

or

$$f^*(x_I) - f^*(x_R) \leq \texttt{mip_integral_gap_abs} * \max(1, |f^*(x_I)|), \tag{6.33}$$

where `mip_integral_gap_abs` and `mip_integral_gap_rel` are typically small positive numbers.

Since these termination conditions assume that the continuous subproblems are solved to global optimality and KNITRO only finds local solutions of nonconvex, continuous optimization problems, they are only reliable when solving convex, mixed integer problems. The integrality gap $f^*(x_I) - f^*(x_R)$ should be non-negative although it may become slightly negative from roundoff error, or if the continuous subproblems are not solved to sufficient accuracy. If the integrality gap becomes largely negative, this may be an indication that the model is nonconvex, in which case KNITRO may not converge to the optimal solution, and will be unable to verify optimality (even if it claims otherwise).

7 KNITRO output and solution information

This section provides information on understanding the KNITRO output and accessing solution information.

7.1 Understanding KNITRO output for continuous problems

If `outlev=0` then all printing of output is suppressed. If `outlev` is positive, then KNITRO prints information about the solution of your optimization problem either to standard output (`outmode=screen`), to a file named knitro.log (`outmode=file`), or to both (`outmode=both`). The option `outdir` controls the directory where output files are created (if any are) and the option `outappend` controls whether output is appended to existing files. See Section 5 for more details.

This section describes KNITRO outputs at various levels for continuous problems. We examine the output that results from running examples/C/callback2_static to solve problemHS15.c.

Display of Nondefault Options:

KNITRO first prints the banner displaying the Ziena license type and version of KNITRO that is installed. It then lists all user options which are different from their default values (see Section 5 for the default user option settings). If nothing is listed in this section then it must be that all user options are set to their default values. Lastly, KNITRO prints messages that describe how it resolved user options that were set to `AUTOMATIC` values. For example, if option `algorithm=auto`, then KNITRO prints the algorithm that it chooses.

```
=====================================
       Commercial Ziena License
             KNITRO 7.0.0
       Ziena Optimization, Inc.
=====================================

outlev:       6
KNITRO changing algorithm from AUTO to 1.
KNITRO changing bar_murule from AUTO to 1.
KNITRO changing bar_initpt from AUTO to 2.
KNITRO changing bar_penaltyrule from AUTO to 1.
KNITRO changing bar_penaltycons from AUTO to 1.
```

In the example above, it is indicated that we are using a more verbose output level `outlev=6` instead of the default value `outlev=2`. KNITRO chose algorithm 1 (Interior/Direct), and then determined four other options related to the algorithm.

Display of Problem Characteristics:

KNITRO next prints a summary description of the problem characteristics including the number and type of variables and constraints and the number of nonzero elements in the Jacobian matrix and Hessian matrix (if providing the exact Hessian).

```
Problem Characteristics
-----------------------
```

```
Objective goal:  Minimize
Number of variables:                    2
    bounded below:                      0
    bounded above:                      1
    bounded below and above:            0
    fixed:                              0
    free:                               1
Number of constraints:                  2
    linear equalities:                  0
    nonlinear equalities:               0
    linear inequalities:                0
    nonlinear inequalities:             2
    range:                              0
Number of nonzeros in Jacobian:         4
Number of nonzeros in Hessian:          3
```

Display of Iteration Information:

Next, if `outlev` is greater than 2, KNITRO prints columns of data reflecting detailed information about individual iterations during the solution process. An iteration is defined as a step which generates a new solution estimate (i.e., a successful step).

If `outlev=2`, summary data is printed every 10 iterations, and on the final iteration. If `outlev=3`, summary data is printed every iteration. If `outlev=4`, the most verbose iteration information is printed every iteration.

Iter	fCount	Objective	FeasError	OptError	||Step||	CGits
0	1	9.090000e+02	3.000e+00			
1	2	7.989784e+02	2.878e+00	9.096e+01	6.566e-02	0
2	3	4.232342e+02	2.554e+00	5.828e+01	2.356e-01	0
3	4	1.457686e+01	9.532e-01	3.088e+00	1.909e+00	0
4	9	1.235269e+02	7.860e-01	3.818e+00	7.601e-01	5
5	10	3.993788e+02	3.022e-02	1.795e+01	1.186e+00	0
6	11	3.924231e+02	2.924e-02	1.038e+01	1.856e-02	0
7	12	3.158787e+02	0.000e+00	6.905e-02	2.373e-01	0
8	13	3.075530e+02	0.000e+00	6.888e-03	2.255e-02	0
9	14	3.065107e+02	0.000e+00	6.397e-05	2.699e-03	0
10	15	3.065001e+02	0.000e+00	4.457e-07	2.714e-05	0

The meaning of each column is described below.

`Iter:` Iteration number.

`fCount:` The cumulative number of function evalutions. (This information is only printed if `outlev` is greater than 3).

`Objective:` Gives the value of the objective function at the current iterate.

`FeasError:` Gives a measure of the feasibility violation at the current iterate (see Section 6).

`OptError:` Gives a measure of the violation of the Karush-Kuhn-Tucker (KKT) (first-order) optimality conditions (not including feasibility) at the current iterate (see Section 6).

`||Step||:` The 2-norm length of the step (i.e., the distance between the new iterate and the previous iterate).

`CGits:` The number of Projected Conjugate Gradient (CG) iterations required to compute the step.

Display of Termination Status:

At the end of the run a termination message is printed indicating whether or not the optimal solution was found and if not, why KNITRO stopped. The termination message typically starts with the word "`EXIT:`". If KNITRO was successful in satisfying the termination test (see Section 6), the message will look as follows:

```
EXIT: Locally optimal solution found.
```

See the appendix for a list of possible termination messages and a description of their meaning and the corresponding value returned by `KTR_solve()`.

Display of Final Statistics:

Following the termination message, a summary of some final statistics on the run are printed. Both relative and absolute error values are printed.

```
Final Statistics
----------------
Final objective value               =   3.06500096351765e+02
Final feasibility error (abs / rel) =   0.00e+00 / 0.00e+00
Final optimality error  (abs / rel) =   4.46e-07 / 3.06e-08
# of iterations                     =         10
# of CG iterations                  =          5
# of function evaluations           =         15
# of gradient evaluations           =         11
# of Hessian evaluations            =         10
Total program time (secs)           =       0.00136 (     0.000 CPU time)
Time spent in evaluations (sec)     =       0.00012
```

Display of Solution Vector and Constraints:

If `outlev` equals 5 or 6, the values of the solution vector are printed after the final statistics. If `outlev` equals 6, the final constraint values are also printed, and the values of the Lagrange multipliers (or dual variables) are printed next to their corresponding constraint or bound.

```
Constraint Vector                   Lagrange Multipliers
-----------------                   --------------------
c[       0] =   1.00000006873e+00,  lambda[       0] =  -7.00000062964e+02
c[       1] =   4.50000096310e+00,  lambda[       1] =  -1.07240081095e-05

Solution Vector
```

```
---------------
x[       0] =   4.99999972449e-01,   lambda[       2] =   7.27764067199e+01
x[       1] =   2.00000024766e+00,   lambda[       3] =   0.00000000000e+00

================================================================================
```

KNITRO can produce additional information which may be useful in debugging or analyzing performance. If `outlev` is positive and `debug=1`, then multiple files named `kdbg_*.log` are created which contain detailed information on performance. If `outlev` is positive and `debug=2`, then KNITRO prints information useful for debugging program execution. The information produced by `debug` is primarily intended for developers, and should not be used in a production setting.

Users can generate a file containing iterates and/or solution points with option `newpoint`. The output file is called knitro_newpoint.log. See Section 5 for details.

7.2 Understanding KNITRO output for discrete problems

If `outlev=0` then all printing of output is suppressed. If `outlev` is positive, then KNITRO prints information about the solution of your optimization problem either to standard output (`outmode=screen`), to a file named knitro.log (`outmode=file`), or to both (`outmode=both`). The option `outdir` controls the directory where output files are created (if any are) and the option `outappend` controls whether output is appended to existing files. When `outlev` is positive, the options `mip_outlevel`, `mip_debug`, `mip_outinterval` and `mip_outsub` control the amount and type of MIP output generated as described below. See Section 5 for more details.

This section describes KNITRO outputs at various levels for discrete or mixed integer problems. We examine the output that results from running examples/C/callbackMINLP_static to solve problemMINLP.c.

KNITRO first prints the banner displaying the Ziena license type and version of KNITRO that is installed. It then lists all user options which are different from their default values (see Section 5 for the default user option settings). If nothing is listed in this section then it must be that all user options are set to their default values. Lastly, KNITRO prints messages that describe how it resolved user options that were set to `AUTOMATIC` values. For example, if option `mip_branchrule=auto`, then KNITRO prints the branching rule that it chooses.

```
=====================================
         Commercial Ziena License
             KNITRO 7.0.0
       Ziena Optimization, Inc.
=====================================

mip_method:           1
mip_outinterval:      1
KNITRO changing mip_rootalg from AUTO to 1.
KNITRO changing mip_lpalg from AUTO to 3.
KNITRO changing mip_branchrule from AUTO to 2.
KNITRO changing mip_selectrule from AUTO to 2.
KNITRO changing mip_rounding from AUTO to 3.
KNITRO changing mip_heuristic from AUTO to 1.
KNITRO changing mip_pseudoinit from AUTO to 1.
```

In the example above, it is indicated that we are using `mip_method=1` which is the standard branch and bound method (see Section 5), and that we are printing output information at every node since `mip_outinterval=1`. It then determined seven other options related to the MIP method.

Display of Problem Characteristics:

KNITRO next prints a summary description of the problem characteristics including the number and type of variables and constraints and the number of nonzero elements in the Jacobian matrix and Hessian matrix (if providing the exact Hessian).

If no initial point is provided by the user, KNITRO indicates that it is computing one. KNITRO also prints the results of any MIP preprocessing to detect special structure and indicates which MIP method it is using.

```
Problem Characteristics
-----------------------
Objective goal:  Minimize
Number of variables:                     6
    bounded below:                       0
    bounded above:                       0
    bounded below and above:             6
    fixed:                               0
    free:                                0
Number of binary variables:              3
Number of integer variables:             0
Number of constraints:                   6
    linear equalities:                   0
    nonlinear equalities:                0
    linear inequalities:                 4
    nonlinear inequalities:              2
    range:                               0
Number of nonzeros in Jacobian:         16
Number of nonzeros in Hessian:           3

No start point provided -- KNITRO computing one.

KNITRO detected 1 GUB constraints
KNITRO derived 0 knapsack covers after examining 3 constraints
KNITRO solving root node relaxation
KNITRO MIP using Branch and Bound method
```

Display of Node Information:

Next, if `mip_outlevel=1`, KNITRO prints columns of data reflecting detailed information about individual nodes during the solution process. The frequency of this node information is controlled by the `mip_outinterval` parameter. For example, if `mip_outinterval=100`, this node information is printed only for every 100th node (printing output less frequently may save significant CPU time

in some cases). In the example below, `mip_outinterval=1`, so information about every node is printed.

```
    Node    Left    Iinf     Objective         Best relaxatn  Best incumbent
   ------  ------  ------  --------------     --------------  --------------
       1       0       2    7.592845e-01       7.592845e-01
       2       1       1    5.171320e+00       7.592845e-01
*      2       1                         r                    7.671320e+00
*      3       2       0    6.009759e+00  f    5.171320e+00    6.009759e+00
       4       1            1.000000e+01 pr    5.171320e+00    6.009759e+00
       5       0            7.092732e+00 pr    6.009759e+00    6.009759e+00
```

The meaning of each column is described below.

`Node:` The node number. If an integer feasible point was found at a given node, then it is marked with a `*`

`Left:` The current number of active nodes left in the branch and bound tree.

`Iinf:` The number of integer infeasible variables at the current node solution.

`Objective:` Gives the value of the objective function at the solution of the relaxed subproblem solved at the current node. If the subproblem was infeasible or failed, this is indicated. Additional symbols may be printed at some nodes if the node was pruned (`pr`), integer feasible (`f`), or an integer feasible point was found through rounding (`r`).

`Best relaxatn:` The value of the current best relaxation (lower bound on the solution if minimizing) (see Section 6.2).

`Best incumbent:` The value of the current best integer feasible point (upper bound on the solution if minimizing) (see Section 6.2).

Display of Termination Status:

At the end of the run a termination message is printed indicating whether or not the optimal solution was found and if not, why KNITRO stopped. The termination message typically starts with the word "`EXIT:`". If KNITRO was successful in satisfying the termination test (see Section 6.2), the message will look as follows:

```
EXIT: Optimal solution found.
```

See the appendix for a list of possible termination messages and a description of their meaning and the corresponding value returned by `KTR_mip_solve()`.

Display of Final Statistics:

Following the termination message, a summary of some final statistics on the run are printed.

```
Final Statistics for MIP
------------------------
Final objective value               =   6.00975890892825e+00
Final integrality gap (abs / rel)   =   0.00e+00 /   0.00e+00 ( 0.00%)
# of nodes processed                =        5
# of subproblems solved             =        6
Total program time (secs)           =       0.09930 (     0.099 CPU time)
Time spent in evaluations (secs)    =       0.00117
```

Display of Solution Vector and Constraints:
If `outlev` equals 5 or 6, the values of the solution vector are printed after the final statistics.

```
Solution Vector
---------------
x[       0] =   1.30097589089e+00
x[       1] =   0.00000000000e+00
x[       2] =   1.00000000000e+00
x[       3] =   0.00000000000e+00  (binary variable)
x[       4] =   1.00000000000e+00  (binary variable)
x[       5] =   0.00000000000e+00  (binary variable)

================================================================================
```

KNITRO can produce additional information which may be useful in debugging or analyzing MIP performance. If `outlev` is positive and `mip_debug=1`, then the file named `kdbg_mip.log` is created which contains detailed information on the MIP performance. In addition, if `mip_outsub=1`, this file will contain extensive output for each subproblem solve in the MIP solution process. The information produced by `mip_debug` is primarily intended for developers, and should not be used in a production setting.

7.3 Accessing solution information

Important solution information from KNITRO is either made available as output from the call to `KTR_solve()`/`KTR_mip_solve()` or can be retrieved through special function calls.

The `KTR_solve()`/`KTR_mip_solve()` functions (see Section 4) return the final value of the objective function in `obj`, the final (primal) solution vector in the array `x` and the final values of the Lagrange multipliers (or dual variables) in the array `lambda`. The solution status code is given by the return value from `KTR_solve()`/`KTR_mip_solve()`.

In addition, information related to the final statistics can be retrieved through the following function calls:

```
int KTR_get_number_FC_evals (const KTR_context_ptr  kc);
```

This function call returns the number of function evaluations requested by `KTR_solve()` or `KTR_mip_solve()`. It returns a negative number if there is a problem with `kc`.

```
int KTR_get_number_GA_evals (const KTR_context_ptr  kc);
```

This function call returns the number of gradient evaluations requested by KTR_solve() or KTR_mip_solve(). It returns a negative number if there is a problem with kc.

```
int KTR_get_number_H_evals (const KTR_context_ptr  kc);
```

This function call returns the number of Hessian evaluations requested by KTR_solve() or KTR_mip_solve(). It returns a negative number if there is a problem with kc.

```
int KTR_get_number_HV_evals (const KTR_context_ptr  kc);
```

This function call returns the number of Hessian-vector products requested by KTR_solve() or KTR_mip_solve(). It returns a negative number if there is a problem with kc.

Continuous problems

```
int KTR_get_number_iters (const KTR_context_ptr  kc);
```

This function returns the number of iterations made by KTR_solve(). It returns a negative number if there is a problem with kc.

```
double KTR_get_abs_feas_error (const KTR_context_ptr  kc);
```

This function returns the absolute feasibility error at the solution. See 6.1 for a detailed definition of this quantity. It returns a negative number if there is a problem with kc.

```
double KTR_get_rel_feas_error (const KTR_context_ptr  kc);
```

This function returns the relative feasibility error at the solution. See 6.1 for a detailed definition of this quantity. It returns a negative number if there is a problem with kc.

```
double KTR_get_abs_opt_error (const KTR_context_ptr  kc);
```

This function returns the absolute optimality error at the solution. See 6.1 for a detailed definition of this quantity. It returns a negative number if there is a problem with kc.

```
double KTR_get_rel_opt_error (const KTR_context_ptr  kc);
```

This function returns the relative optimality error at the solution. See 6.1 for a detailed definition of this quantity. It returns a negative number if there is a problem with kc.

Discrete or mixed integer problems

```
int KTR_get_mip_num_nodes (const KTR_context_ptr  kc);
```

This function returns the number of nodes processed in the MIP solve made by KTR_mip_solve(). It returns a negative number if there is a problem with kc.

```
int KTR_get_mip_num_solves (const KTR_context_ptr  kc);
```

This function returns the number of continuous subproblems processed in the MIP solve made by KTR_mip_solve(). It returns a negative number if there is a problem with kc.

```
double KTR_get_mip_abs_gap (const KTR_context_ptr  kc);
```

This function returns the final absolute integrality gap in the MIP solve made by `KTR_mip_solve()`. See 6.2 for a detailed definition of this quantity. It returns `KTR_INFBOUND` if no incumbent (i.e., integer feasible) point was found. It returns termination code `KTR_RC_BAD_KCPTR` if there is a problem with `kc`.

```
double KTR_get_mip_rel_gap (const KTR_context_ptr  kc);
```

This function returns the final relative integrality gap in the MIP solve made by `KTR_mip_solve()`. See 6.2 for a detailed definition of this quantity. It returns `KTR_INFBOUND` if no incumbent (i.e., integer feasible) point was found. It returns termination code `KTR_RC_BAD_KCPTR` if there is a problem with `kc`.

```
double  KTR_get_mip_incumbent_obj (const KTR_context_ptr  kc);
```

This function returns the objective value of the MIP incumbent solution. It returns `KTR_INFBOUND` if no incumbent (i.e., integer feasible) point has been found. It returns termination code `KTR_RC_BAD_KCPTR` if there is a problem with `kc`.

```
int  KTR_get_mip_incumbent_x (const KTR_context_ptr kc,
                              double * const x);
```

If an incumbent solution exists, it is returned in `x`, otherwise `x` is not modified. The function return value is 1 if an incumbent solution exists and the function was successful in retrieving it; 0 if no incumbent solution exists yet; or a negative number if there was an error in the function call.

```
double  KTR_get_mip_relaxation_bnd (const KTR_context_ptr  kc);
```

This function returns the value of the current MIP relaxation bound. It returns termination code `KTR_RC_BAD_KCPTR` if there is a problem with `kc`.

```
double  KTR_get_mip_lastnode_obj (const KTR_context_ptr  kc);
```

This function returns the objective value of the most recently solved MIP node subproblem. It returns termination code `KTR_RC_BAD_KCPTR` if there is a problem with `kc`.

8 Algorithm options

8.1 Automatic

KNITRO provides three different algorithms for solving problems. See Section 1.2 for an overview of the methods. By default, KNITRO automatically tries to choose the best algorithm for a given problem based on problem characteristics.

We strongly encourage you to experiment with all the algorithms as it is difficult to predict which one will work best on any particular problem.

8.2 Interior/Direct

This algorithm often works best, and will automatically switch to Interior/CG if the direct step is suspected to be of poor quality, or if negative curvature is detected. Interior/Direct is recommended if the Hessian of the Lagrangian is ill-conditioned. The Interior/CG method in this case will often take an excessive number of conjugate gradient iterations. It may also work best when there are dependent or degenerate constraints. Choose this algorithm by setting user option `algorithm=1`.

We encourage you to experiment with different values of the `bar_murule` option when using the Interior/Direct or Interior/CG algorithm. It is difficult to predict which update rule will work best on a problem.

NOTE: Since the Interior/Direct algorithm in KNITRO requires the explicit storage of a Hessian matrix, this algorithm only works with Hessian options (`hessopt`) 1, 2, 3, or 6 (see Section 9.2). It may not be used with Hessian options 4 or 5, which do not supply a full Hessian matrix. The Interior/Direct algorithm may be used with the `bar_feasible` option.

8.3 Interior/CG

This algorithm is well-suited to large problems because it avoids forming and factorizing the Hessian matrix. Interior/CG is recommended if the Hessian is large and/or dense. It works with all Hessian options, and with the `bar_feasible` option. Choose this algorithm by setting user option `algorithm=2`.

We encourage you to experiment with different values of the `bar_murule` option when using the Interior/Direct or Interior/CG algorithm. It is difficult to predict which update rule will work best on a problem.

8.4 Active Set

This algorithm is fundamentally different from interior-point methods. The method is efficient and robust for small and medium-scale problems, but is typically less efficient than the Interior/Direct and Interior/CG algorithms on large-scale problems (many thousands of variables and constraints). Active Set is recommended when "warm starting" (i.e., when the user can provide a good initial solution estimate, for example, when solving a sequence of closely related problems). This algorithm is also best at rapid detection of infeasible problems. Choose this algorithm by setting user option `algorithm=3`.

NOTE: The `bar_feasible` option (see Section 9.3) is not available for use with the Active Set algorithm. The method works with all Hessian options.

9 Other KNITRO special features

This section describes in more detail some of the most important features of KNITRO. It provides some guidance on which features to use so that KNITRO runs most efficiently for the problem at hand.

9.1 First derivative and gradient check options

The default version of KNITRO assumes that the user can provide exact first derivatives to compute the objective function gradient and constraint gradients. It is *highly* recommended that the user provide exact first derivatives if at all possible, since using first derivative approximations may seriously degrade the performance of the code and the likelihood of converging to a solution. However, if this is not possible the following first derivative approximation options may be used.

Forward finite-differences
This option uses a forward finite-difference approximation of the objective and constraint gradients. The cost of computing this approximation is n function evaluations where n is the number of variables. The option is invoked by choosing user option `gradopt=2` (see Section 5).

Centered finite-differences
This option uses a centered finite-difference approximation of the objective and constraint gradients. The cost of computing this approximation is $2n$ function evaluations where n is the number of variables. The option is invoked by choosing user option `gradopt=3` (see Section 5). The centered finite-difference approximation is often more accurate than the forward finite-difference approximation; however, it is more expensive to compute if the cost of evaluating a function is high.

Gradient Checks
If the user supplies a routine for computing exact gradients, KNITRO can easily check them against finite-difference gradient approximations. To do this, modify your application and replace the call to `KTR_solve()`/`KTR_mip_solve()` with `KTR_check_first_ders()`, then run the application. KNITRO will call the user routine for exact gradients, compute finite-difference approximations, and print any differences that exceed a given threshold. KNITRO also checks that the sparse constraint Jacobian has all nonzero elements defined. The check can be made with forward or centered differences. A sample driver is provided in examples/C/checkDersExample.c. Small differences between exact and finite-difference approximations are to be expected (see comments in examples/C/checkDersExample.c). It is best to check the gradient at different points, and to avoid points where partial derivatives happen to equal zero.

9.2 Second derivative options

The default version of KNITRO assumes that the application can provide exact second derivatives to compute the Hessian of the Lagrangian function. If the application is able to do so and the cost of computing the second derivatives is not overly expensive, it is highly recommended to provide exact second derivatives. However, KNITRO also offers other options which are described in detail below.

(Dense) Quasi-Newton BFGS
The quasi-Newton BFGS option uses gradient information to compute a symmetric, *positive-definite*

approximation to the Hessian matrix. Typically this method requires more iterations to converge than the exact Hessian version. However, since it is only computing gradients rather than Hessians, this approach may be more efficient in some cases. This option stores a *dense* quasi-Newton Hessian approximation so it is only recommended for small to medium problems ($n < 1000$). The quasi-Newton BFGS option is chosen by setting user option `hessopt=2`.

(Dense) Quasi-Newton SR1
As with the BFGS approach, the quasi-Newton SR1 approach builds an approximate Hessian using gradient information. However, unlike the BFGS approximation, the SR1 Hessian approximation is not restricted to be positive-definite. Therefore the quasi-Newton SR1 approximation may be a better approach, compared to the BFGS method, if there is a lot of negative curvature in the problem since it may be able to maintain a better approximation to the true Hessian in this case. The quasi-Newton SR1 approximation maintains a *dense* Hessian approximation and so is only recommended for small to medium problems ($n < 1000$). The quasi-Newton SR1 option is chosen by setting user option `hessopt=3`.

Finite-difference Hessian-vector product option
If the problem is large and gradient evaluations are not a dominant cost, then KNITRO can internally compute Hessian-vector products using finite-differences. Each Hessian-vector product in this case requires one additional gradient evaluation. This option is chosen by setting user option `hessopt=4`. The option is only recommended if the exact gradients are provided.

NOTE: This option may not be used when `algorithm=1`.

Exact Hessian-vector products
In some cases the application may prefer to provide exact Hessian-vector products, but not the full Hessian (for instance, if the problem has a large, dense Hessian). The application must provide a routine which, given a vector v stored in `hessVector`, computes the Hessian-vector product, Hv, and returns the result in `hessVector`. This option is chosen by setting user option `hessopt=5`.

NOTE: This option may not be used when `algorithm=1`.

Limited-memory Quasi-Newton BFGS
The limited-memory quasi-Newton BFGS option is similar to the dense quasi-Newton BFGS option described above. However, it is better suited for large-scale problems since, instead of storing a dense Hessian approximation, it stores only a limited number of gradient vectors used to approximate the Hessian. The number of gradient vectors used to approximate the Hessian is controlled by user option `lmsize`.

A larger value of `lmsize` may result in a more accurate, but also more expensive, Hessian approximation. A smaller value may give a less accurate, but faster, Hessian approximation. When using the limited memory BFGS approach it is recommended to experiment with different values of this parameter.

In general, the limited-memory BFGS option requires more iterations to converge than the dense quasi-Newton BFGS approach, but will be much more efficient on large-scale problems. The limited-memory quasi-Newton option is chosen by setting user option `hessopt=6`.

9.3 Feasibility options

KNITRO offers an option `bar_feasible` that can force iterates to stay feasible with respect to *inequality* constraints or can place special emphasis on trying to get feasible.

If `bar_feasible=1` or `bar_feasible=3` KNITRO satisfies inequalities by switching to a *feasible mode* of operation, which alters the manner in which iterates are computed. The option does not enforce feasibility with respect to *equality* constraints, as this would impact performance too much. The theory behind feasible mode is described in [5].

The initial point must satisfy inequalities to a sufficient degree; if not, KNITRO may generate infeasible iterates and does not switch to the feasible mode until a sufficiently feasible point is found. We say *sufficient* satisfaction occurs at a point x if it is true for all inequalities that

$$cl + tol \leq c(x) \leq cu - tol \tag{9.34}$$

The constant $tol > 0$ is determined by the option `bar_feasmodetol`; its default value is `1.0e-4`. Feasible mode becomes active once an iterate x satisfies (9.34) for all inequality constraints. If the initial point satisfies (9.34), then every iterate will be feasible with respect to the inequalities.

KNITRO can also place special emphasis on *getting* feasible (with respect to all constraints) through the option `bar_feasible`. If `bar_feasible=2` or `bar_feasible=3`, KNITRO will first place special emphasis on getting feasible before working on optimality. This option is not always guaranteed to accelerate the finding of a feasible point. However, it may do a better job of obtaining feasibility on difficult problems where the default version struggles.

NOTE: This option can only be used with the Interior/Direct and Interior/CG algorithms.

9.4 Honor bounds mode

In some applications, the user may want to enforce that the initial point and all subsequent iterates satisfy the simple bounds $bl \leq x \leq bu$. For instance, if the objective function or a nonlinear constraint function is undefined at points outside the bounds, then the bounds should be enforced at all times.

By default, KNITRO enforces bounds on the variables only for the initial start point and the final solution (`honorbnds=2`). To enforce satisfaction at all iterates, set `honorbnds=1`. To allow execution from an initial point that violates the bounds, set `honorbnds=0`.

9.5 Crossover

Interior-point (or barrier) methods are a powerful tool for solving large-scale optimization problems. However, one drawback of these methods is that they do not always provide a clear picture of which constraints are active at the solution. In general they return a less exact solution and less exact sensitivity information. For this reason, KNITRO offers a *crossover* feature in which the interior-point method switches to the Active Set method at the interior-point solution estimate, in order to "clean up" the solution and provide more exact sensitivity and active set information.

The crossover procedure is controlled by the `maxcrossit` user option. If this parameter is greater than 0, then KNITRO will attempt to perform `maxcrossit` Active Set crossover iterations after the interior-point method has finished, to see if it can provide a more exact solution. This can be viewed as a form of post-processing. If `maxcrossit` is not positive, then no crossover iterations are attempted.

The crossover procedure will not always succeed in obtaining a more exact solution compared with the interior-point solution. If crossover is unable to improve the solution within `maxcrossit` crossover iterations, then it will restore the interior-point solution estimate and terminate. If `outlev` is greater than one, KNITRO will print a message indicating that it was unable to improve the solution. For example, if `maxcrossit=3`, and the crossover procedure did not succeed, the message will read:

```
Crossover mode unable to improve solution within 3 iterations.
```

In this case, you may want to increase the value of `maxcrossit` and try again. If KNITRO determines that the crossover procedure will not succeed, no matter how many iterations are tried, then a message of the form

```
Crossover mode unable to improve solution.
```

will be printed.

The extra cost of performing crossover is problem dependent. In most small or medium scale problems, the crossover cost is a small fraction of the total solve cost. In these cases it may be worth using the crossover procedure to obtain a more exact solution. On some large scale or difficult degenerate problems, however, the cost of performing crossover may be significant. It is recommended to experiment with this option to see whether improvement in the exactness of the solution is worth the additional cost.

9.6 Multi-start

Nonlinear optimization problems (1.1) are often nonconvex due to the objective function, constraint functions, or both. When this is true, there may be many points that satisfy the local optimality conditions described in Section 6. Default KNITRO behavior is to return the first locally optimal point found. KNITRO offers a simple *multi-start* feature that searches for a better optimal point by restarting KNITRO from different initial points. The feature is enabled by setting `ms_enable=1`.

The multi-start procedure generates new start points by randomly selecting components of x that satisfy lower and upper bounds on the variables. KNITRO finds a local optimum from each start point using the same problem definition and user options. The final solution returned from `KTR_solve()` is the local optimum with the best objective function value if any local optimum have been found. If no local optimum have been found, KNITRO will return the best feasible solution estimate it found. If no feasible solution estimate has been found, KNITRO will return the least infeasible point. If you wish to see details of the local optimization process for each start point, then set option `outlev` to at least 4.

The number of start points tried by multi-start is specified with the option `ms_maxsolves`. By default, KNITRO will try $\min\{200, 10n\}$, where n is the number of variables in the problem. Users may override the default by setting `ms_maxsolves` to a specific value.

The multi-start option is convenient for conducting a simple search for a better solution point. Search time is improved if the variable bounds are made as tight as possible, confining the search to a region where a good solution is likely to be found. The user can restrict the multi-start search region without altering bounds by using the options `ms_maxbndrange` and `ms_startptrange`. The first option applies to variables unbounded in at least one direction (i.e., the upper or lower bound (or both) is infinite) and keeps new start points within a total range equal to the value of `ms_maxbndrange`. The second option applies to all variables and keeps new start points within a total

range equal to the value of `ms_startptrange`, overruling `ms_maxbndrange` if it is a tighter bound. In general, use `ms_startptrange` to limit the multi-start search only if the initial start point supplied by the user is known to be the center of a desired search area. Use `ms_maxbndrange` as a surrogate bound to limit the multi-start search when a variable is unbounded. See Section 5.1 for details.

The `ms_num_to_save` option allows a specific number of distinct feasible points to be saved in a file named knitro_mspoints.log. Each point results from a KNITRO solve from a different starting point, and must satisfy the absolute and relative feasibility tolerances. Different start points may return the same feasible point, and the file contains only distinct points. The option `ms_savetol` determines that two points are distinct if their objectives or any solution components (including Lagrange multipliers) are separated by more than the value of `ms_savetol` using a relative tolerance test. More specifically, two values x and y are considered distinct if

$$|x - y| \geq \max(1, |x|, |y|) * \texttt{ms_savetol}. \tag{9.35}$$

The file stores points in order from best objective to worst. If objectives are the same (as defined by `ms_savetol`), then points are ordered from smallest feasibility error to largest. The file can be read manually, but conforms to a fixed property/value format for machine reading.

Instead of using multi-start to search for a global solution, a user may want to use multi-start as a mechanism for finding any locally optimal or feasible solution estimate of a nonconvex problem and terminate as soon as one such point is found. The `ms_terminate` option, provides the user more control over when to terminate the multi-start procedure. If `ms_terminate=optimal` the multi-start procedure will stop as soon as the first locally optimal solution is found or after `ms_maxsolves` – whichever comes first. If `ms_terminate=feasible` the multi-start procedure will instead stop as soon as the first feasible solution estimate is found or after `ms_maxsolves` – whichever comes first. If `ms_terminate=maxsolves`, it will only terminate after `ms_maxsolves`.

In most cases the user would like to obtain the *global optimum* to (1.1); that is, the local optimum with the very best objective function value. KNITRO cannot guarantee that multi-start will find the global optimum. In general, the global optimum can only be found with special knowledge of the objective and constraint functions; for example, the functions may need to be bounded by other piece-wise convex functions. KNITRO executes with very little information about functional form. Although no guarantee can be made, the probability of finding a better local solution improves if more start points are tried. See Section 10.6 for more discussion.

9.7 Reverse communication mode for invoking KNITRO

The reverse communication mode of KNITRO returns control to the user at the driver level whenever a function, gradient, or Hessian evaluation is needed, making it easy to embed the KNITRO solver into an application. In addition, this mode allows applications to monitor or stop the progress of the KNITRO solver after each iteration, based on any criteria the user desires.

If the return value from `KTR_solve()`/`KTR_mip_solve()` is 0 or negative, the optimization is finished (see Appendix A). If the return value is positive, KNITRO requires that some task be performed by the user at the driver level before re-entering `KTR_solve()`/`KTR_mip_solve()`. Referring to the optimization problem formulation given in (4.4), the action to take for possible positive return values are:

`KTR_RC_EVALFC (1):` Evaluate functions $f(x)$ and $c(x)$ and re-enter `KTR_solve()` or `KTR_mip_solve()`.

`KTR_RC_EVALGA (2):` Evaluate gradient $\nabla f(x)$ and the constraint Jacobian matrix and re-enter `KTR_solve()` or `KTR_mip_solve()`.

`KTR_RC_EVALH (3):` Evaluate the Hessian $H(x, \lambda)$ and re-enter `KTR_solve()` or `KTR_mip_solve()`.

`KTR_RC_EVALHV (7):` Evaluate the Hessian $H(x, \lambda)$ times a vector and re-enter `KTR_solve()` or `KTR_mip_solve()`.

`KTR_RC_NEWPOINT (6):` KNITRO has just computed a new solution estimate, and the function and gradient values are up-to-date. The user may provide routines to perform some task. Then the application must re-enter `KTR_solve()` so that KNITRO can begin a new iteration. `KTR_RC_NEWPOINT` is only returned if user option `newpoint=user` (and is only valid for continuous problems).

Section 4.3 describes an example program that uses the reverse communications mode.

9.8 Callback mode for invoking KNITRO

The callback mode of KNITRO requires the user to supply several function pointers that KNITRO calls when it needs new function, gradient or Hessian values, or to execute a user-provided newpoint routine. For convenience, every one of these callback routines takes the same list of arguments. If your callback requires additional parameters, you are encouraged to create a structure containing them and pass its address as the `userParams` pointer. KNITRO does not modify or dereference the `userParams` pointer, so it is safe to use for this purpose. Section 4.2 describes an example program that uses the callback mode.

The C language prototype for the KNITRO callback function is defined in knitro.h:

```
typedef int KTR_callback (const int             evalRequestCode,
                          const int             n,
                          const int             m,
                          const int             nnzJ,
                          const int             nnzH,
                          const double * const  x,
                          const double * const  lambda,
                                double * const  obj,
                                double * const  c,
                                double * const  objGrad,
                                double * const  jac,
                                double * const  hessian,
                                double * const  hessVector,
                                void   *        userParams);
```

The callback functions for evaluating the functions, gradients and Hessian or for performing some newpoint task, are set as described below. Each user callback routine should return an `int` value of 0 if successful, or a negative value to indicate that an error occurred during execution of the user-provided function. Section 4.2 describes example program that uses the callback mode.

```
/* This callback should modify "obj" and "c". */
int KTR_set_func_callback (KTR_context_ptr kc, KTR_callback * func);

/* This callback should modify "objGrad" and "jac". */
int KTR_set_grad_callback (KTR_context_ptr kc, KTR_callback * func);

/* This callback should modify "hessian" or "hessVector",
   depending on the value of "evalRequestCode". */
int KTR_set_hess_callback (KTR_context_ptr kc, KTR_callback * func);

/* This callback should modify nothing. */
int KTR_set_newpoint_callback (KTR_context_ptr kc, KTR_callback * func);
```

NOTE: To enable "newpoint" callbacks, set `newpoint=user`. These should only be used for continuous problems.

KNITRO also provides a special callback function for output printing. By default KNITRO prints to stdout or a `knitro.log` file, as determined by the `outmode` option. Alternatively, the user can define a callback function to handle all output. This callback function can be set as shown below.

```
int KTR_set_puts_callback (KTR_context_ptr kc, KTR_puts * puts_func);
```

The prototype for the KNITRO callback function used for handling output is:

```
typedef int KTR_puts (char * str, void * user);
```

10 Special problem classes

This section describes specializations in KNITRO to deal with particular classes of optimization problems. We also provide guidance on how to best set user options and model your problem to get the best performance out of KNITRO for particular types of problems.

10.1 Linear programming problems (LPs)

A linear program (LP) is an optimization problem where the objective function and all the constraint functions are linear.

KNITRO has built in specializations for efficiently solving LPs. However, KNITRO is unable to automatically detect whether or not a problem is an LP. In order for KNITRO to detect that a problem is an LP, you must specify this by setting the value of `objType` to `KTR_OBJTYPE_LINEAR` and all values of the array `cType` to `KTR_CONTYPE_LINEAR` in the function call to `KTR_init_problem()` (see Section 4). If this is not done, KNITRO will not apply special treatment to the LP and will typically be less efficient in solving the LP.

10.2 Quadratic programming problems (QPs)

A quadratic program (QP) is an optimization problem where the objective function is quadratic and all the constraint functions are linear.

KNITRO has built in specializations for efficiently solving QPs. However, KNITRO is unable to automatically detect whether or not a problem is a QP. In order for KNITRO to detect that a problem is a QP, you must specify this by setting the value of `objType` to `KTR_OBJTYPE_QUADRATIC` and all values of the array `cType` to `KTR_CONTYPE_LINEAR` in the function call to `KTR_init_problem()` (see Section 4). If this is not done, KNITRO will not apply special treatment to the QP and will typically be less efficient in solving the QP.

Typically, these specialization will only help on convex QPs.

10.3 Systems of nonlinear equations

KNITRO is effective at solving systems of nonlinear equations. To solve a square system of nonlinear equations using KNITRO one should specify the nonlinear equations as equality constraints (i.e., $c^L = c^U$ in (1.1*b*)), and specify the objective function (1.1*a*) as zero (i.e., $f(x) = 0$).

If KNITRO is converging to a stationary point for which the nonlinear equations are not satisfied, the multi-start option described in Section 9.6, may help in finding a solution by trying different starting points.

10.4 Least squares problems

There are two ways of using KNITRO for solving problems in which the objective function is a sum of squares of the form

$$f(x) = \frac{1}{2} \sum_{j=1}^{q} r_j(x)^2.$$

If the value of the objective function at the solution is not close to zero (the large residual case), the least squares structure of f can be ignored and the problem can be solved as any other optimization problem. Any of the KNITRO options can be used.

On the other hand, if the optimal objective function value is expected to be small (small residual case) then KNITRO can implement the Gauss-Newton or Levenberg-Marquardt methods which only require first derivatives of the residual functions, $r_j(x)$, and yet converge rapidly. To do so, the user need only define the Hessian of f to be

$$\nabla^2 f(x) = J(x)^T J(x),$$

where

$$J(x) = \left[\frac{\partial r_j}{\partial x_i}\right] \begin{array}{l} j = 1, 2, \ldots, q \\ i = 1, 2, \ldots, n \end{array}.$$

The actual Hessian is given by

$$\nabla^2 f(x) = J(x)^T J(x) + \sum_{j=1}^{q} r_j(x) \nabla^2 r_j(x);$$

the Gauss-Newton and Levenberg-Marquardt approaches consist of ignoring the last term in the Hessian.

KNITRO will behave like a Gauss-Newton method by setting `algorithm=1`, and will be very similar to the classical Levenberg-Marquardt method when `algorithm=2`. For a discussion of these methods see, for example, [10].

10.5 Complementarity constraints (MPCCs)

A mathematical program with complementarity (or equilibrium) constraints (also know as an MPCC or MPEC) is an optimization problem which contains a particular type of constraint referred to as a complementarity constraint. A complementarity constraint is a constraint which enforces that two variables are *complementary* to each other. The variables x_1 and x_2 are complementary if the following conditions hold

$$x_1 \times x_2 = 0, \quad x_1 \geq 0, \quad x_2 \geq 0. \tag{10.36}$$

The condition above is sometimes expressed more compactly as

$$0 \leq x_1 \quad \perp \quad x_2 \geq 0.$$

One could also have more generally, that a particular constraint is complementary to another constraint or a constraint is complementary to a variable. However, by adding slack variables, a complementarity constraint can always be expressed as two variables complementary to each other, and KNITRO requires that you express complementarity constraints in this form. For example, if you have two constraints $c_1(x)$ and $c_2(x)$ which are complementary

$$c_1(x) \times c_2(x) = 0, \quad c_1(x) \geq 0, \quad c_2(x) \geq 0,$$

you can re-write this as two equality constraints and two complementary variables, s_1 and s_2 as follows:

$$s_1 = c_1(x) \tag{10.37}$$

$$s_2 = c_2(x) \tag{10.38}$$

$$s_1 \times s_2 = 0, \quad s_1 \geq 0, \quad s_2 \geq 0. \tag{10.39}$$

Intuitively, a complementarity constraint is a way to model a constraint which is combinatorial in nature since, for example, the conditions in (10.36) imply that either x_1 or x_2 must be 0 (both may be 0 as well). Without special care, these type of constraints may cause problems for nonlinear optimization solvers because problems which contain these types of constraints fail to satisfy constraint qualifications which are often assumed in the theory and design of algorithms for nonlinear optimization. For this, reason we provide a special interface in KNITRO for specifying complementarity constraints. In this way, KNITRO can recognize these constraints and apply some special care to them internally.

Complementarity constraints can be specified in KNITRO through a call to the function `KTR_addcompcons()` which has the following prototype and argument list.

Prototype:

```
int KTR_addcompcons(KTR_context_ptr kc,
                    int numCompConstraints,
                    int * indexList1,
                    int * indexList2);
```

Arguments:

`KTR_context *kc`: is a pointer to a structure which holds all the relevant information about a particular problem instance.

`int numCompConstraints`: is a scalar specifying the number of complementarity constraints to be added to the problem (i.e., the number of pairs of variables which are complementary to each other).

`int *indexList1`: is an array of length `numCompConstraints` specifying the variable indices for the first set of variables in the pairs of complementary variables.

`int *indexList2`: is an array of length `numCompConstraints` specifying the variable indices for the second set of variables in the pairs of complementary variables.

The call to `KTR_addcompcons()` must occur after the call to `KTR_init_problem()`, but before the first call to `KTR_solve()`. Below we provide a simple example of how to define the KNITRO data structures to specify a problem which includes complementarity constraints.

Example: [1]

Assume we want to solve the following MPEC with KNITRO.

$$\begin{aligned}
\underset{x}{\text{minimize}} \quad & f(x) = (x_0 - 5)^2 + (2x_1 + 1)^2 && \text{(10.40a)} \\
\text{subject to} \quad & c_0(x) = 2(x_1 - 1) - 1.5x_0 + x_2 - 0.5x_3 + x_4 = 0 && \text{(10.40b)} \\
& c_1(x) = 3x_0 - x_1 - 3 \geq 0 && \text{(10.40c)} \\
& c_2(x) = -x_0 + 0.5x_1 + 4 \geq 0 && \text{(10.40d)}
\end{aligned}$$

[1] An MPEC from J.F. Bard, Convex two-level optimization, *Mathematical Programming* 40(1), 15-27, 1988.

$$c_3(x) = -x_0 - x_1 + 7 \geq 0 \tag{10.40e}$$
$$x_i \geq 0, i = 0..4 \tag{10.40f}$$
$$c_1(x)x_2 = 0 \tag{10.40g}$$
$$c_2(x)x_3 = 0 \tag{10.40h}$$
$$c_3(x)x_4 = 0. \tag{10.40i}$$

It is easy to see that the last 3 constraints (along with the corresponding non-negativity conditions) represent complementarity constraints. Expressing this in compact notation, we have:

$$\underset{x}{\text{minimize}} \quad f(x) = (x_0 - 5)^2 + (2x_1 + 1)^2 \tag{10.41a}$$
$$\text{subject to} \quad c_0(x) = 0 \tag{10.41b}$$
$$0 \leq c_1(x) \perp x_2 \geq 0 \tag{10.41c}$$
$$0 \leq c_2(x) \perp x_3 \geq 0 \tag{10.41d}$$
$$0 \leq c_3(x) \perp x_4 \geq 0 \tag{10.41e}$$
$$x_0 \geq 0, x_1 \geq 0. \tag{10.41f}$$

Since KNITRO requires that complementarity constraints be written as two variables complementary to each other, we must introduce slack variables x_5, x_6, x_7 and re-write problem (10.40) as

$$\underset{x}{\text{minimize}} \quad f(x) = (x_0 - 5)^2 + (2x_1 + 1)^2 \tag{10.42a}$$
$$\text{subject to} \quad c_0(x) = 2(x_1 - 1) - 1.5x_0 + x_2 - 0.5x_3 + x_4 = 0 \tag{10.42b}$$
$$\tilde{c_1}(x) = 3x_0 - x_1 - 3 - x_5 = 0 \tag{10.42c}$$
$$\tilde{c_2}(x) = -x_0 + 0.5x_1 + 4 - x_6 = 0 \tag{10.42d}$$
$$\tilde{c_3}(x) = -x_0 - x_1 + 7 - x_7 = 0 \tag{10.42e}$$
$$x_i \geq 0, i = 0..7 \tag{10.42f}$$
$$x_2 \perp x_5 \tag{10.42g}$$
$$x_3 \perp x_6 \tag{10.42h}$$
$$x_4 \perp x_7. \tag{10.42i}$$

Now that the problem is in a form suitable for KNITRO, we define the problem for KNITRO by using `c`, `cLoBnds`, and `cUpBnds` for (10.42*b*)-(10.42*e*), and `xLoBnds`, `xUpBnds` for (10.42*f*) to specify the normal constraints and bounds in the usual way for KNITRO. We use `indexList1`, `indexList2` and the `KTR_addcompcons()` function call to specify the complementarity constraints (10.42*g*)-(10.42*i*). These arrays are specified as follows for (10.42).

```
n = 8;                  /* number of variables */
m = 4;                  /* number of regular constraints */
numCompConstraints = 3; /* number of complementarity constraints */

c[0] = 2*(x[1]-1) - 1.5*x[0] + x[2] - 0.5*x[3] + x[4];
c[1] = 3*x[0] - x[1] - 3 -x[5];
c[2] = -x[0] + 0.5*x[1] + 4 -x[6];
```

```
c[3] = -x[0] - x[1] + 7 - x[7];

cLoBnds[0] = 0;  cUpBnds[0] = 0;
cLoBnds[1] = 0;  cUpBnds[1] = 0;
cLoBnds[2] = 0;  cUpBnds[2] = 0;
cLoBnds[3] = 0;  cUpBnds[3] = 0;

xLoBnds[0] = 0;  xUpBnds[0] = KTR_INFBOUND;
xLoBnds[1] = 0;  xUpBnds[1] = KTR_INFBOUND;
xLoBnds[2] = 0;  xUpBnds[2] = KTR_INFBOUND;
xLoBnds[3] = 0;  xUpBnds[3] = KTR_INFBOUND;
xLoBnds[4] = 0;  xUpBnds[4] = KTR_INFBOUND;
xLoBnds[5] = 0;  xUpBnds[5] = KTR_INFBOUND;
xLoBnds[6] = 0;  xUpBnds[6] = KTR_INFBOUND;
xLoBnds[7] = 0;  xUpBnds[7] = KTR_INFBOUND;

indexList1[0] = 2;   indexList2[0] = 5;
indexList1[1] = 3;   indexList2[1] = 6;
indexList1[2] = 4;   indexList2[2] = 7;
```

NOTE: Variables which are specified as complementary through the special `KTR_addcompcons()` functions should be specified to have a lower bound of 0 through the KNITRO lower bound array `xLoBnds`.

When using KNITRO through a particular modeling language, only some modeling languages allow for the identification of complementarity constraints. If a modeling language does not allow you to specifically identify and express complementarity constraints, then these constraints must be formulated as regular constraints and KNITRO will not perform any specializations.

10.6 Global optimization

KNITRO is designed for finding locally optimal solutions of continuous optimization problems. A local solution is a feasible point at which the objective function value at that point is as good or better than at any "nearby" feasible point. A globally optimal solution is one which gives the best (i.e., lowest if minimizing) value of the objective function out of all feasible points. If the problem is *convex* all locally optimal solutions are also globally optimal solutions. The ability to guarantee convergence to the global solution on large-scale *nonconvex* problems is a nearly impossible task on most problems unless the problem has some special structure or the person modeling the problem has some special knowledge about the geometry of the problem. Even finding local solutions to large-scale, nonlinear, nonconvex problems is quite challenging.

Although KNITRO is unable to guarantee convergence to global solutions it does provide a *multi-start* heuristic which attempts to find multiple local solutions in the hopes of locating the global solution. See Section 9.6 for information on trying to find the globally optimal solution using the KNITRO multi-start feature.

10.7 Mixed integer programming (MIP)

KNITRO provides tools for solving optimization models (both linear and nonlinear) with binary or integer variables. The KNITRO mixed integer programming (MIP) code offers two algorithms for mixed-integer nonlinear programming (MINLP). The first is a nonlinear branch and bound method and the second implements the hybrid Quesada-Grossman method for convex MINLP.

The KNITRO MINLP code is designed for convex mixed integer programming and is a heuristic for nonconvex problems. The MIP code also handles mixed integer linear programs (MILP) of moderate size. A MIP problem is defined and solved via the callable library interface using the API functions `KTR_mip_init_problem()` and `KTR_mip_solve()` (see Section 4). The KNITRO MIP tools do not currently handle special ordered sets (SOS's) or semi-continuous variables. Examples for solving a MINLP problem using the C and Java interfaces are provided with the distribution in the `examples` directory.

Many user options are provided for the MIP features to tune performance, including options for branching, node selection, rounding and heuristics for finding integer feasible points. User options specific to the MIP tools begin with `mip_` (see Section 5). It is recommended to experiment with several of these options as they often can make a significant difference in performance. In particular, if finding any integer feasible point is your highest priority, you should set the `mip_heuristic` option to search for an integer feasible point before beginning the branch and bound procedure (by default no heuristics are applied).

The MIP features are new in KNITRO 6.0 and may not be available for every interface immediately.

11 Tips and troubleshooting

This section describes special tips for getting the best performance out of the KNITRO solver and dealing with particular performance issues that may arise. Refer to Section 5.1 for more details and explanations regarding user option settings referenced below.

11.1 Option tuning for efficiency

KNITRO has a great number and variety of user option settings and although it tries to choose the best settings by default, often significant performance improvements can be realized by choosing some non-default option settings. The list below seeks to highlight some options that are particularly important, and to provide some basic guidance on how to set some options based on the particular characteristics of your model.

- The most important user option is the choice of which continuous nonlinear optimization algorithm to use which is specified by the `algorithm` option. Please try all three values as it is often difficult to predict which one will work best. In particular the Active Set (`active`) algorithm may often work best for small problems, problems whose only constraints are simple bounds on the variables, or linear programs. The interior-point algorithms are generally preferable for large-scale problems.

- Perhaps the second most important user option setting is the `hessopt` user option that specifies which Hessian (or Hessian approximation) technique to use. If you (or the modeling language) are not providing the `exact` Hessian to KNITRO, then you should experiment with different values here. More details on this option are provide in Section 9.2.

- One of the most important user options for the interior-point algorithms is the `bar_murule` option. It is recommended to experiment with different values for this user option if you are using one of the interior-point solvers in KNITRO.

- If you are using the Interior/Direct (`direct`) algorithm and it seems to be taking a large number of conjugate gradient (CG) steps (as evidenced by a non-zero value under the `CGits` output column header on many iterations), then you should try a small value for the `bar_directinterval` user option (e.g., 0-2). This option will try to prevent KNITRO from taking an excessive number of CG steps. Additionally, if there are solver iterations where KNITRO slows down because it is taking a very large number of CG iterations, you can try enforcing a maximum limit on the number of CG iterations per algorithm iteration using the `maxcgit` user option.

- The `linsolver` option can make a big difference in performance for some problems. For small problems (particularly small problems with dense Jacobian and Hessian matrices), it is recommended to try the `qr` setting, while for large problems, it is recommended to try the `hybrid`, `ma27` and `ma57` settings to see which is fastest. When using either the `hybrid`, `qr` or `ma57` setting for the `linsolver` option it is *highly* recommended to use the Intel MKL BLAS (`blasoption=1`) provided with KNITRO or some other optimized BLAS as this can result in significant speedups compared to the internal KNITRO BLAS (`blasoption=0`).

- When solving mixed integer problems (MIPs), if KNITRO is struglling to find an integer feasible point, then you should try different values for the `mip_heuristic` option which will try to find a integer feasible point before beginning the branch and bound process. Other important

MIP options that can significantly impact the performance of KNITRO are the `mip_method`, `mip_branchrule`, and `mip_selectrule` user options, as well as the `algorithm` option which will determine the KNITRO algorithm to use to solve the nonlinear, continuous subproblems generated during the branch and bound process.

11.2 Memory issues

If you receive a KNITRO termination message indicating that there was not enough memory on your computer to solve the problem, or if your problem appears to be running very slow because it is using nearly all of the available memory on your computer system, the following are some recommendations to try to reduce the amount of memory used by KNITRO.

- Experiment with different algorithms. Typically the Interior/Direct algorithm is chosen by default and uses the most memory. The Interior/CG and Active Set algorithms usually use much less memory. In particular if the Hessian matrix is large and dense and using most of the memory, then the Interior/CG method may offer big savings in memory. If the constraint Jacobian matrix is large and dense and using most of the memory, then the Active Set algorithm may use much less memory on your problem.

- If much of the memory usage seems to come from the Hessian matrix, then you should try different Hessian options via the `hessopt` user option. In particular `hessopt` settings `finite_diff`, `product`, and `lbfgs` use the least amount of memory.

- Try different linear solver options in KNITRO via the `linsolver` user option. Sometimes even if your problem definition (e.g. Hessian and Jacobian matrix) can be easily stored in memory, the sparse linear system solvers inside KNITRO may require a lot of extra memory to perform and store matrix factorizations. If your problem size is relatively small you can try `linsolver` setting `qr`. For large problems you should try both `ma27` and `ma57` settings as one or the other may use significantly less memory. In addition, using a smaller `pivot` user option value may reduce the amount of memory needed for the linear solver.

11.3 Infeasibility

KNITRO is a solver for finding *local* solutions to general nonlinear, possibly nonconvex problems. Just as KNITRO may converge to a local solution that is not the global solution, it is also possible for a nonlinear optimization solver to converge to a *locally infeasible point* or *infeasible stationary point* on nonconvex problems. That is, even if the users' model is feasible, a nonlinear solver can converge to a point where the model is locally infeasible. At this point, a move in any direction will increase some measure of infeasibility and thus a local solver cannot make any further progress from such a point. Just as only a global optimization solver can guarantee that it will locate the globally optimal solution, only a global solver can also avoid the possibility of converging to these locally infeasible points.

If your problem is nonconvex and the KNITRO termination message indicates that it has converged to an infeasible point, then you should try running KNITRO again from a different starting point (preferably one close to the feasible region). Alternatively, you can use the KNITRO multi-start feature (see Section 9.6) which will automatically try to run KNITRO several times from different starting points, to try to avoid getting stuck at locally infeasible points.

If you are using one of the interior-point algorithms in KNITRO, and KNITRO is struggling to find a feasible point, you can try different settings for the `bar_feasible` user option to place special emphasis on obtaining feasibility.

References

[1] R. H. Byrd, J.-Ch. Gilbert, and J. Nocedal. A trust region method based on interior point techniques for nonlinear programming. *Mathematical Programming*, 89(1):149–185, 2000.

[2] R. H. Byrd, N. I. M. Gould, J. Nocedal, and R. A. Waltz. On the convergence of successive linear-quadratic programming algorithms. *SIAM Journal on Optimization*, 16(2):471–489, 2006.

[3] R. H. Byrd, N. I. M. Gould, J. Nocedal, and R. A. Waltz. An algorithm for nonlinear optimization using linear programming and equality constrained subproblems. *Mathematical Programming, Series B*, 100(1):27–48, 2004.

[4] R. H. Byrd, M. E. Hribar, and J. Nocedal. An interior point algorithm for large scale nonlinear programming. *SIAM Journal on Optimization*, 9(4):877–900, 1999.

[5] R. H. Byrd, J. Nocedal, and R. A. Waltz. Feasible interior methods using slacks for nonlinear optimization. *Computational Optimization and Applications*, 26(1):35–61, 2003.

[6] R. H. Byrd, J. Nocedal, and R.A. Waltz. KNITRO: An integrated package for nonlinear optimization. In G. di Pillo and M. Roma, editors, *Large-Scale Nonlinear Optimization*, pages 35–59. Springer, 2006.

[7] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming, 2nd Ed.*. Brooks/Cole – Thomson Learning, 2003.

[8] Harwell Subroutine Library. *A catalogue of subroutines (HSL 2002)*. AEA Technology, Harwell, Oxfordshire, England, 2002.

[9] Hock, W. and Schittkowski, K. *Test Examples for Nonlinear Programming Codes*, volume 187 of *Lecture Notes in Economics and Mathematical Systems*. Springer-Verlag, 1981.

[10] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, 1999.

[11] R. A. Waltz, J. L. Morales, J. Nocedal, and D. Orban. An interior algorithm for nonlinear optimization that combines line search and trust region steps. *Mathematical Programming A*, 107(3):391–408, 2006.

Appendix A

Solution Status Codes

The solution status return codes are organized as follows.

0:	The final solution satisfies the termination conditions for verifying optimality.
-100 to -199:	A feasible approximate solution was found.
-200 to -299:	The code terminated at an infeasible point.
-300:	The problem was determined to be unbounded.
-400 to -499:	The code terminated because it reached a pre-defined limit.
-500 to -599:	The code terminated with an input error or some non-standard error.

A more detailed description of individual return codes and their corresponding termination messages is provided below.

`0 (KTR_RC_OPTIMAL):`

`Locally optimal solution found.`

KNITRO found a locally optimal point which satisfies the stopping criterion (see Section 6 for more detail on how this is defined). If the problem is convex (for example, a linear program), then this point corresponds to a globally optimal solution.

`-100 (KTR_RC_NEAR_OPT):`

```
Primal feasible solution estimate cannot be improved.  It appears to be
optimal, but desired accuracy in dual feasibility could not be achieved.
```

No more progress can be made, but the stopping tests are close to being satisfied (within a factor of 100) and so the current approximate solution is believed to be optimal.

`-101 (KTR_RC_FEAS_XTOL):`

```
Primal feasible solution; terminate because the relative change in
solution estimate < xtol.  Decrease xtol to try for more accuracy.
```

The optimization terminated because the relative change in the solution estimate is less than that specified by the parameter `xtol`. To try to get more accuracy one may decrease `xtol`. If `xtol` is very small already, it is an indication that no more significant progress can be made. It's possible the approximate feasible solution is optimal, but perhaps the stopping tests cannot be satisfied because of degeneracy, ill-conditioning or bad scaling.

`-102 (KTR_RC_FEAS_NO_IMPROVE):`

```
Primal feasible solution estimate cannot be improved; desired accuracy
in dual feasibility could not be achieved.
```

No further progress can be made. It's possible the approximate feasible solution is optimal, but perhaps the stopping tests cannot be satisfied because of degeneracy, ill-conditioning or bad scaling.

`-200 (KTR_RC_INFEASIBLE):`

```
Convergence to an infeasible point.  Problem may be locally
infeasible.  If problem is believed to be feasible, try multistart
to search for feasible points.
```

The algorithm has converged to an infeasible point from which it cannot further decrease the infeasibility measure. This happens when the problem is infeasible, but may also occur on occasion for feasible problems with nonlinear constraints or badly scaled problems. It is recommended to try various initial points with the multi-start feature described in Section 9.6. If this occurs for a variety of initial points, it is likely the problem is infeasible.

`-201 (KTR_RC_INFEAS_XTOL):`

```
Terminate at infeasible point because the relative change in solution
estimate < xtol.  Decrease xtol to try for more accuracy.
```

The optimization terminated because the relative change in the solution estimate is less than that specified by the parameter `xtol`. To try to find a feasible point one may decrease `xtol`. If `xtol` is very small already, it is an indication that no more significant progress can be made. It is recommended to try various initial points with the multi-start feature described in Section 9.6. If this occurs for a variety of initial points, it is likely the problem is infeasible.

`-202 (KTR_RC_INFEAS_NO_IMPROVE):`

```
Current infeasible solution estimate cannot be improved.  Problem may
be badly scaled or perhaps infeasible.  If problem is believed to be
feasible, try multistart to search for feasible points.
```

No further progress can be made. It is recommended to try various initial points with the multi-start feature described in Section 9.6. If this occurs for a variety of initial points, it is likely the problem is infeasible.

`-203 (KTR_RC_INFEAS_MULTISTART):`

```
MULTISTART: No primal feasible point found.
```

The multi-start feature was unable to find a feasible point. If the problem is believed to be feasible, then increase the number of initial points tried in the multi-start feature and also perhaps increase the range from which random initial points are chosen. See Section 9.6 for more details about multi-start and Section 5.1 for various multi-start user options.

`-300 (KTR_RC_UNBOUNDED):`

```
Problem appears to be unbounded.  Iterate is feasible and objective
magnitude > objrange.
```

The objective function appears to be decreasing without bound, while satisfying the constraints. If the problem really is bounded, increase the size of the parameter `objrange` to avoid terminating with this message.

`-400 (KTR_RC_ITER_LIMIT):`

```
Iteration limit reached.
```

The iteration limit was reached before being able to satisfy the required stopping criteria. The iteration limit can be increased through the user option `maxit`. See Section 5.1.

`-401 (KTR_RC_TIME_LIMIT):`

`Time limit reached.`

The time limit was reached before being able to satisfy the required stopping criteria. The time limit can be increased through the user options `maxtime_cpu` and `maxtime_real`. See Section 5.1.

`-403 (KTR_RC_MIP_EXH):`

`All nodes have been explored.`

The MIP optimality gap has not been reduced below the specified threshold, but there are no more nodes to explore in the branch and bound tree. If the problem is convex, this could occur if the gap tolerance is difficult to meet because of bad scaling or roundoff errors, or there was a failure at one or more of the subproblem nodes. This might also occur if the problem is nonconvex. In this case, KNITRO terminates and returns the best integer feasible point found.

`-404 (KTR_RC_MIP_FEAS_TERM):`

`Terminating at first integer feasible point.`

KNITRO has found an integer feasible point and is terminating because the user option `mip_terminate = feasible`. See Section 5.1.

`-405 (KTR_RC_MIP_SOLVE_LIMIT):`

`Subproblem solve limit reached.`

The MIP subproblem solve limit was reached before being able to satisfy the optimality gap tolerance. The subproblem solve limit can be increased through the user option `mip_maxsolves`. See Section 5.1.

`-406 (KTR_RC_MIP_NODE_LIMIT):`

`Node limit reached.`

The MIP node limit was reached before being able to satisfy the optimality gap tolerance. The node limit can be increased through the user option `mip_maxnodes`. See Section 5.1.

`-500 (KTR_RC_CALLBACK_ERR):`

`Callback function error.`

This termination value indicates that an error (i.e., negative return value) occurred in a user provided callback routine.

`-501 (KTR_RC_LP_SOLVER_ERR):`

`LP solver error.`

This termination value indicates that an unrecoverable error occurred in the LP solver used in the active-set algorithm preventing the optimization from continuing.

`-502 (KTR_RC_EVAL_ERR):`

`Evaluation error.`

This termination value indicates that an evaluation error occurred (e.g., divide by 0, taking the square root of a negative number), preventing the optimization from continuing.

`-503 (KTR_RC_OUT_OF_MEMORY):`

`Not enough memory available to solve problem.`

This termination value indicates that there was not enough memory available to solve the problem.

`-505 to -600:`

Termination values in this range imply some input error or other non-standard failure. If `outlev>0` details of this error will be printed to standard output or the file knitro.log depending on the value of `outmode`.

Appendix B
List of Output Files

knitro.log

This is the standard output from KNITRO. The file is created if `outmode=file` or `outmode=both`. See Section 7 for an explanation of the contents.

knitro_mspoints.log

This file contains a set of feasible points found by multi-start, each distinct, in order of best to worst. The file is created if `ms_enable=yes` and `ms_num_to_save` is greater than zero. See Section 9.6 for more information.

knitro_newpoint.log

This file contains a set of iterates generated by KNITRO. It is created if `newpoint` equals `saveone` or `saveall`.

kdbg_barrierIP.log
kdbg_directIP.log
kdbg_normalIP.log
kdbg_profileIP.log
kdbg_stepIP.log
kdbg_summIP.log
kdbg_tangIP.log

These files contain detailed debug information. The files are created if `debug=problem` and either barrier method (Interior/Direct or Interior/CG) executes. The kdbg_directIP.log file is created only for the Interior/Direct method.

kdbg_actsetAS.log
kdbg_eqpAS.log
kdbg_lpAS.log
kdbg_profileAS.log
kdbg_stepAS.log
kdbg_summAS.log

These files contain detailed debug information. The files are created if `debug=problem` and the Active Set method executes.

kdbg_mip.log

This file contains detailed debug information. The file is created if `mip_debug=all` and one of the MIP methods executes.